Compiler Optimization for Reducing Leakage Power in Multithread BSP Programs

WEN-LI SHIH, National Tsing Hua University
YI-PING YOU, National Chiao Tung University
CHUNG-WEN HUANG and JENQ KUEN LEE, National Tsing Hua University

Multithread programming is widely adopted in novel embedded system applications due to its high performance and flexibility. This article addresses compiler optimization for reducing the power consumption of multithread programs. A traditional compiler employs energy management techniques that analyze component usage in control-flow graphs with a focus on single-thread programs. In this environment the leakage power can be controlled by inserting on and off instructions based on component usage information generated by flow equations. However, these methods cannot be directly extended to a multithread environment due to concurrent execution issues.

This article presents a multithread power-gating framework composed of *multithread power-gating analysis* (MTPGA) and *predicated power-gating* (PPG) energy management mechanisms for reducing the leakage power when executing multithread programs on *simultaneous multithreading* (SMT) machines. Our multithread programming model is based on hierarchical bulk-synchronous parallel (BSP) models. Based on a multithread component analysis with dataflow equations, our MTPGA framework estimates the energy usage of multithread programs and inserts PPG operations as power controls for energy management. We performed experiments by incorporating our power optimization framework into SUIF compiler tools and by simulating the energy consumption with a post-estimated SMT simulator based on Wattch toolkits. The experimental results show that the total energy consumption of a system with PPG support and our power optimization method is reduced by an average of 10.09% for BSP programs relative to a system without a power-gating mechanism on leakage contribution set to 30%; and the total energy consumption is reduced by an average of 4.27% on leakage contribution set to 10%. The results demonstrate our mechanisms are effective in reducing the leakage energy of BSP multithread programs.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent*; *distributed*; *parallel languages*; D.3.4 [**Programming Languages**]: Processors—*Compiler*; *optimization*

General Terms: Design, Language

Additional Key Words and Phrases: Compilers for low power, leakage power reduction, power-gating mechanisms, multithreading

ACM Reference Format:
Wen-Li Shih, Yi-Ping You, Chung-Wen Huang, and Jenq Kuen Lee. 2014. Compiler optimization for reducing leakage power in multithread BSP programs. ACM Trans. Des. Autom. Electron. Syst. 20, 1, Article 9 (November 2014), 34 pages.
DOI: http://dx.doi.org/10.1145/2668119

This work is supported in part by Ministry of Science and Technology (under grant no. 103-2220-E-007-019) and Ministry of Economic Affairs (under grant no. 103-EC-17-A-02-S1-202) in Taiwan.
Author's addresses: W.-L. Shih, Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan; Y.-P. You, Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan; C.-W. Huang and J. K. Lee (corresponding author), Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan; email: jklee@cs.nthu.edu.tw.

DOI: http://dx.doi.org/10.1145/2668119

1. INTRODUCTION

Approaches for minimizing power dissipation can be applied at the algorithmic, compiler, architectural, logic, and circuit levels [Chandrakasan et al. 1992]. Aspects relative to combining architecture design and software arrangement at the instruction level have been addressed with the aim of reducing power consumption [Bellas et al. 2000; Chang and Pedram 1995; Horowitz et al. 1994; Lee et al. 1997, 2003, 2013; Su and Despain 1995; Tiwari et al. 1997, 1998]. Major efforts in power optimization include dynamic and leakage power optimization. Works in dynamic power optimization include utilizing the value locality of registers [Chang and Pedram 1995], scheduling VLIW (very long instruction word) instructions to reduce the power consumption on the instruction bus [Lee et al. 2003], reducing instruction encoding to reduce code size and power consumption [Lee et al. 2013], and gating the clock to reduce workloads [Horowitz et al. 1994; Tiwari et al. 1997, 1998]. Compiler code for reducing leakage power can employ power gating [Kao and Chandrakasan 2000; Butts and Sohi 2000; Hu et al. 2004]. Various studies have attempted to reduce the leakage power using integrated architectures and compiler-based power gating mechanisms [Dropsho et al. 2002; Yang et al. 2002; You et al. 2002, 2007; Rele et al. 2002; Zhang et al. 2003; Li and Xue 2004]. These approaches involve compilers inserting instructions into programs to shut down and wake up components as appropriate, based on a dataflow analysis or a profiling analysis. The power analysis and instruction insertion are further integrated into trace-based binary translation [Li and Xue 2004]. The *Sink-N-Hoist* framework [You et al. 2005, 2007] has been used to reduce the number of power-gating instructions generated by compilers. However, these power-gating control frameworks are only applicable to single-thread programs, and care is needed in multithread programs since some of the threads might share the same hardware resources. Turning resources on and off requires careful consideration of cases where multiple threads are present. Herein, we extend previous work to deal with the case of multithread systems in a *bulk-synchronous parallel* (BSP) model.

The BSP model, proposed by Valiant [1990], is designed to bridge between theory and practice of parallel computations. The BSP model structures multiple processors with local memory and a global barrier synchronous mechanism. Threads processed by processors are separated by synchronous points, called supersteps, that form the basic unit of the BSP model. A superstep consists of a computation phase and a communication phase, allowing processors to compute data in local memory until encountering a global synchronous point in the computation phase and synchronizing local data with each other in the communication phase. The algorithm complexity of parallel programs can then be analyzed in the BSP model by considering both locality and parallelism issues. The BSP model works well for a family of parallel applications in which the tasks are balanced. However, global barrier synchronization was found inflexible in the practice [McColl 1996], which promoted proposals for several enhanced BSP models presenting hierarchical groupings. NestStep [Keßler 2000] is a programming language for the BSP model that adopts nested parallelism with support for virtual shared memory. The H-BSP model [Cha and Lee 2001] splits processors into groups and dynamically runs BSP programs within each group in a bulk-synchronous fashion, while the multicore BSP [Valiant 2008, 2011] provides hierarchical multicore environments with independent communication costs. In the present study we adopted the concept of hierarchical BSP models [Keßler 2000; Cha and Lee 2001; Torre and Kruskal 1996] as the basis for a power reduction framework for use in parallel programming.

Several methods have been proposed for analyzing the concurrency of multithread programs. *May-happen-in-parallel* (MHP) analysis computes which statements may be executed concurrently in a multithread program [Callahan and Sublok 1989;

Duesterwald and Soffa 1991; Masticola and Ryder 1993; Naumovich and Avrunin 1998; Naumovich et al. 1999; Li and Verbrugge 2004; Barik 2005]. The problem of precisely computing all pairs of statements that may execute in parallel is undecidable [Ramalingam 2000]; however, it was proved that the problem is NP-complete if we assume that all control paths are executable [Taylor 1983]. The general approach involves using a dataflow framework to compute a conservative estimate of MHP information.

This article presents a multithread power-gating (MTPG) framework, composed of MTPG Analysis (MTPGA) and predicated power-gating (PPG) energy management mechanisms for reducing leakage power when executing multithread programs on *simultaneous multithreading* (SMT) machines. SMT is a widely adopted processor technique that allows multithread programs to utilize functional units more efficiently by fetching and executing instructions from multiple threads at the same time. Our multithread programming model is based on hierarchical BSP models. We propose using *thread fragment concurrency analysis* (TFCA) to analyze MHP information among threads and MTPGA to report the component usages shared by multiple threads in hierarchical BSP models. TFCA reports the concurrency of threads, which allows power-gating candidates to be classified into those used by multiple threads and those used by a single thread. A conventional power-gating optimization framework [You et al. 2005, 2007] can be employed for candidates used by a single thread, with the compiler inserting instructions into the program to shut down and wake up components as appropriate. For candidates used concurrently by different threads, PPG instructions are adopted to turn components on and off as appropriate. Based on the TFCA, our MTPGA framework estimates the energy usage of multithread programs with our proposed cost model and inserts a pair of predicated power-on and predicated power-off operations at those positions where a power-gating candidate is first activated and last deactivated within a thread.

To our best knowledge, this is the first work to attempt to devise an analysis scheme for reducing leakage power in multithread programs. We performed experiments by incorporating TFCA and MTPGA into SUIF compiler tools and by simulating the energy consumption with a post-estimated SMT simulator based on Wattch toolkits. Our preliminary experimental results on a system with leakage contribution set to 30% show that the total energy consumption of a system with PPG support and our power optimization method is reduced by an average of 10.09% for BSP programs converted from the OpenCL kernel and by up to 10.49% for D-BSP programs relative to the system without a power-gating mechanism, and is reduced by an average of 4.27% for BSP programs and by up to 6.68% for D-BSP programs on a system with leakage contribution set to 10%, demonstrating our mechanisms effective in reducing the leakage power in hierarchical BSP multithread environments.

The remainder of the article is organized as follows. Section 2 gives a motivating example for the problem addressed by our study. Section 3 presents the technical rationale of our work, first presenting the PPG instruction and architectures, and then summarizing our compilation flow. Section 4 presents the method of TFCA for hierarchical BSP programs while Section 5 presents our MTPGA compiler framework for power optimizations. Section 6 presents the experimental results, discussion is given in Section 7, and conclusions are drawn in Section 8.

2. MOTIVATION

A system might be equipped with a power-gating mechanism to activate and deactivate components in order to reduce the leakage current [Goodacre 2011]. In such systems, programmers or compilers should analyze the behavior of programs, investigate component utilization based on execution sequences, and insert power-gating

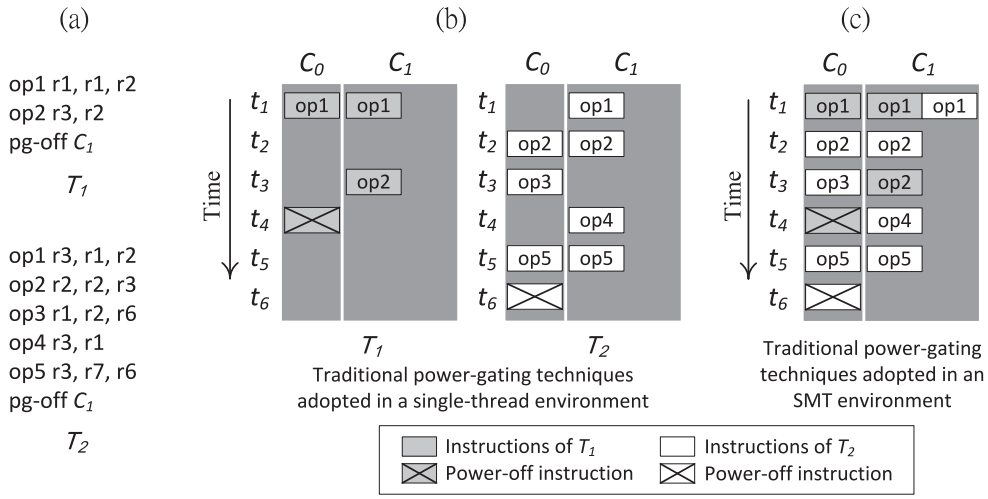

Fig. 1. The traditional power-gating mechanism adopted in a single-thread or SMT environment. Both environments are equipped with two categories of components C_0 and C_1, where C_0 is capable of controlling the power-gating status of C_1: (a) Two code segments of threads T_1 and T_2, where power-gating instructions are inserted by power-gating analysis results for threads T_1 and T_2 individually. Note that $op1$ of T_1 and $op2$ and $op5$ of T_2 demonstrate those cases where instructions might need more than one component (in this case, C_0 and C_1) to complete operation; (b); (c) how the code segments in (a) are executed in a single-thread and SMT environment, respectively. All component usages of instructions for the two threads are labeled as square boxes with corresponding labels, and power-off instructions are labeled as boxes with a cross.

instructions into programs [You et al. 2002, 2006] to ensure that the leakage current is gated appropriately. Traditional compiler analysis algorithms for low power focus on single-thread programs, and the methods cannot be directly applied to multithread programs. We use the example in Figure 1 to illustrate the scenario for motivating the need of new compiler schemes for reducing the power consumption in multithread environments. Assume we have hardware equipped with two categories of functional units, named C_0 and C_1, where C_0 is capable of controlling the power-gating status of C_1, and the hardware is configurable as a single-thread or SMT environment. We first present two pseudocode segments for threads T_1 and T_2 in Figure 1(a), that are analyzed and processed by traditional low-power optimization analysis. Note that $op1$ of T_1 and $op2$ and $op5$ of T_2 demonstrate the cases where instructions might need more than one component (in this case, C_0 and C_1) to complete operation. Traditional sequential analysis of the compiler will yield the component utilization for every instruction. As shown in Figure 1(a), the compiler inserts two power-gating instructions "pg-off C_1" at the end of both code segments because C_1 is no longer used for those segments in subsequent codes. These code segments work smoothly when executed individually in single-thread environments as shown in Figure 1(b). In the figure, all component usages of instructions for the two threads are labeled as square boxes with corresponding labels, and power-off instructions are labeled as boxes with a cross. For thread T_1, after instructions op1 and op2 are executed, the power-off instruction is executed at t_4; hence the system could save leakage energy from idle component C_1. For thread T_2, after five instructions are executed, the power-off instruction is executed at t_6, which turns off component C_1 to stop the leakage current.

However, when the multithread program is executed in an SMT system, the system could concurrently execute threads T_1 and T_2 with shared components C_0 and C_1 as illustrated in Figure 1(c). At time t_4, thread T_1 powers off C_1 because the traditional compiler analysis reports that C_1 will no longer be used in T_1 and a power-off instruction

is inserted. However, T_2 actually still uses C_1 at time t_4 and t_5, which means that powering off C_1 at t_4 will make the system fail if the powered-off components fully rely on power-gating instructions; or the system would pay the penalty associated with executing T_2 at t_4 if the system could internally turn on the components according to the status of instruction queues.

The prior example indicates that the traditional single-thread analyzer cannot be naively applied to the MTPG case, as it will likely break the logic that a unit must be in the active state (i.e., powered on) before being used for processing, since the unit might be powered off by a thread while other concurrent threads are still using or about to use it. Moreover, a unit might be powered on multiple times by a set of concurrent threads. The preceding problems must be appropriately addressed when constructing power-gating controls for multithread programs. This article presents our solution for addressing this issue.

3. TECHNICAL RATIONALE

3.1. PPG Operations

Predicated execution support provides an effective means to eliminate branches from an instruction stream. Predicated or guarded execution refers to the conditional execution of an instruction based on the value of a Boolean source operand, referred to as the predicate [Hsu and Davidson 1986]. Predicated instructions are fetched regardless of their predicate value. Instructions whose predicate is true are executed normally, while those whose predicate is false are nullified and thus prevented from modifying the processor state.

We include the concept of predicated execution in power-gating devices for controlling the power gating of a set of concurrent threads. We combine the predicated executions into three special power-gating operations: predicated power-on, predicated power-off, and initialization operations. The main ideas are: (1) to turn on a component only when it is actually in the off state; (2) to keep track of the number of threads using the component; and (3) to turn off the component when this is the last exit of all threads using this component. Note that these operations must be atomic with respect to each other in order to prevent multiple threads from accessing control at the same time.

—*Initialization operation*. An initialization operation is designed to clean all predicated bits (i.e., pgp_1, pgp_2, ..., pgp_N) and empty all reference counters (i.e., rc_1, rc_2, ..., rc_N) when the processor is starting up.

—*Predicated power-on operation*. The predicated power-on operation takes an explicit operand and two implicit operands to record component usage and conditionally turn on a power-gating candidate. The explicit operand is power-gating candidate C_i, and the implicit operands include predicated bit pgp_i of C_i and a reference counter rc_i of C_i. The operation consists of the following steps:
  (1) power on C_i if pgp_i (i.e., the predicated bit of C_i) is set;
  (2) increase rc_i (i.e., the reference counter of C_i) by 1. The reference counter keeps track of the number of threads that reference the power-gating candidate at this time; and
  (3) unset predicated bit pgp_i.

—*Predicated power-off operation*. The predicated power-off operation also takes an explicit operand C_i and two implicit operands pgp_i and rc_i. Predicated power-off instructions update component usage rc_i and conditionally turn off a power-gating candidate C_i by predicated bit pgp_i. The operation consists of the following steps:
  (1) decrease the reference counter rc_i by 1;
  (2) set predicate bit pgp_i if reference counter rc_i is 0; and
  (3) power off C_i if predicated bit pgp_i is set.

```
lock lc1, lc2, ..., lcN;
pgp1 = pgp2 = ... = pgpN = 0;
rc1 = rc2 = ... = rcN = 0;
unlock lc1, lc2, ..., lcN;
```

(a) initialization

```
        lock lc1;
(pgp1)  poweron C1;
        rc1 = rc1 + 1;
        pgp1 = 0;
        unlock lc1;
```

(b) predicated power-on C_1

```
        lock lc1;
        rc1 = rc1 - 1;
        pgp1 = (rc1 == 0);
(pgp1)  poweroff C1;
        unlock lc1;
```

(c) predicated power-off C_1

Fig. 2. Pseudocode segments to illustrate the specification of PPG operations for a power-gating candidate C_1: (a) Initialization operation for the power-gating mechanism; (b); (c) operations to atomically predicated power-on and predicated power-off C_1, respectively.

Figure 2 illustrates the specification of these PPG instructions in pseudocode segments. Consider a power-gating candidate C_1 in an SMT system with PPG support. Figure 2(a) shows the initialization operation for all PPG operations, and Figures 2(b) and 2(c) show pseudocode segments for predicated power-on and power-off operations for C_1, respectively. To support the atomicity, a lock lc_1 is used before and after the code segments to guarantee that these operations are executed exclusively. For efficiency reasons, hardware circuits should be used to implement this behavior in practice.

3.2. Multithread Power-Gating Framework

Algorithm 1 summarizes our proposed compiler flow of the MTPG framework for BSP models. To generate code with power-gating control in a multithread BSP program, the compiler should compute concurrency information and analyze component usage with respect to concurrent threads. Step 1 of the algorithm applies TFCA to component usages shared by multiple threads in hierarchical BSP models (the details of this algorithm are presented in Section 4); this is the hierarchical BSP version of MHP analysis. In step 2, detailed component usages can be calculated via dataflow equations by referencing *component-activity dataflow analysis* (CADFA) [You et al. 2002, 2006]. Steps 3 and 4 insert PPG instructions according to the information gathered in the previous steps while considering the cost model (Section 5 presents our MTPGA compiler framework for power optimizations). In step 3, MTPGA arranges power-gating control among threads. In step 4, CADFA calculates the detailed component usage with regard to the arrangement of step 3. Steps 5 and 6 further merge the generated power-gating controls into a single compound instruction based on the sink-n-hoist framework [You et al. 2005, 2007]. This is a compiler solution to merge power-gating instructions into a single compound instruction and reduce the number of power-gating instructions issued. Step 5 decides if and where power-gating instructions should be inserted, while

ALGORITHM 1: Multithread Power-Gating Framework

Input: A source program
Output: A program with power-gating control
begin
1 Perform *thread fragment concurrency analysis* for BSP programs
2 Perform *component-activity data-flow analysis* to get detailed component usage
3 Perform *multithread power-gating analysis* to arrange power-gating control among threads
4 Perform *component-activity data-flow analysis* with advise from MTPGA
5 Perform power-gating-instruction scheduling
6 Perform *sink-n-hoist analysis* to merge generated power-gating controls
7 Produce predicated-power-gating instructions and power-gating instruction
end

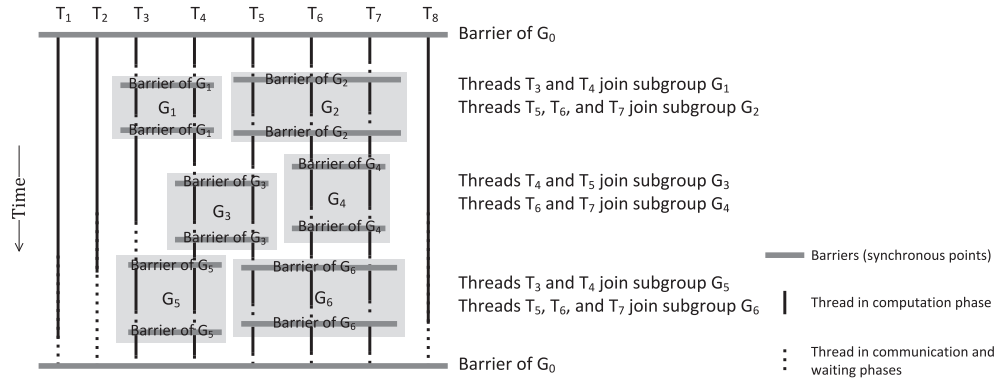

Fig. 3. Illustration of one superstep of a BSP program, where eight threads (T_1 to T_8) are divided into six subgroups (G_1 to G_6). Each subgroup contains a subsuperstep. In a hierarchical BSP program, programmers are allowed to divide threads into groups and the synchronization of threads would be limited in the groups, which form subsupersteps inside the groups. A barrier is a synchronous point of a group in the hierarchical BSP model; therefore all the barriers in program must belong to a specific group as shown in the figure.

step 6 attempts to merge the power-gating instructions with the sink-n-hoist framework. Finally, step 7 produces the power-control assembly codes.

4. TFCA FOR BSP PROGRAMS

This section presents the concurrency analysis method for BSP programs. We consider hierarchical BSP models with a fixed number of threads. Operations of BSP programs are assumed well structured and correctly maintained by programmers, and the scheduling of threads is assumed explicit and correctly maintained by programmers or a static scheduler. Figure 3 presents an example for a superstep of the hierarchical BSP model, in which vertical black lines indicate threads and horizontal gray bars indicate barriers. Eight individual threads and two barriers form the superstep, where the eight threads join and are divided into six groups. In a hierarchical BSP program, programmers are allowed to divide threads into groups and the synchronization of threads would be limited in the groups, which form subsupersteps inside groups. A barrier is a synchronous point of a group in a hierarchical BSP model; therefore, all the barriers in a program must belong to a specific group. The threads in each group are synchronized by barriers belonging to the group, which form subsupersteps inside the groups.

The threads do not have a constant relationship. Computing the concurrency between threads actually involves considering the relation between threads that are present during a specific period, which are indicated by a set of neighboring nodes in the control-flow graph (CFG), denoted by a *thread fragment*. We calculate the thread concurrency in a superstep of a group rather than of the entire BSP program. Since every superstep is executed sequentially, solving the thread concurrency of all supersteps will solve the thread concurrency of the BSP program. This analysis is performed by first constructing a *thread fragment graph* (TFG) that represents the relationships in a superstep, and then computing *lineal thread fragments* and the *may-happen-in-parallel regions* (MHP regions) that represent thread fragments that have a lineal relationship and thread fragments that may happen in parallel, respectively.

4.1. Thread Fragment Graph

The relationships between thread fragments in a superstep are abstracted into a directed graph named the TFG, in which a node represents a thread fragment and an edge represents the control flow. A TFG might be constructed with a single CFG or multiple

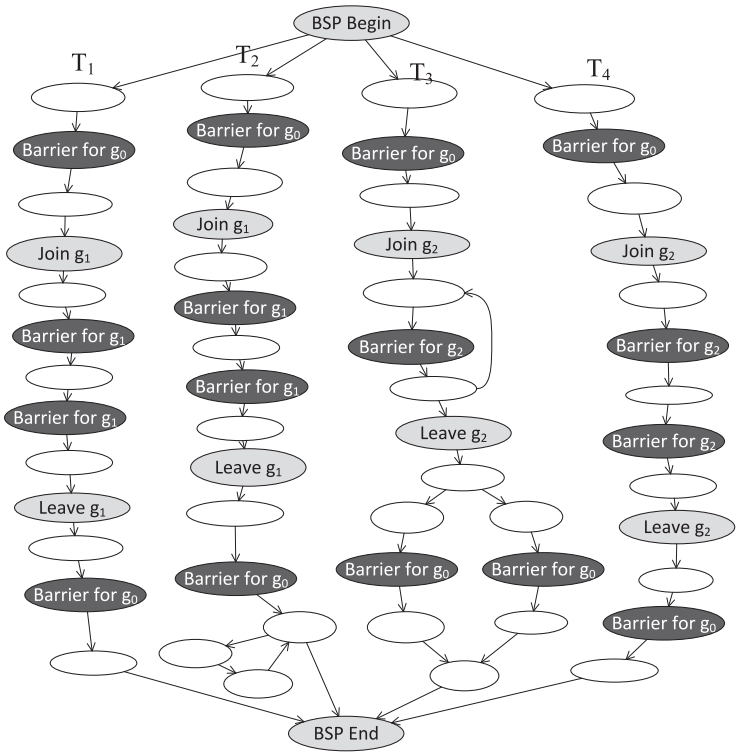

Fig. 4. Hierarchical BSP program presented in a CFG, where four threads (T_1 to T_4) are divided into four supersteps by barriers. In the second superstep, four threads are further grouped into two groups (g_1 to g_2). Each subgroup has its own supersteps.

CFGs, depending on the adopted programming model. For a *single-program multiple-data* programming model, a multithread program is a single executable file and is executed heterogeneously by conditional branches of unique *thread identification*; a TFG for a single-program multiple-data program is thus constructed from a CFG where certain control paths are recognized as thread operations. For a *multiple-program multiple-data* programming model, a multithread program is composed of multiple individual executable files that are executed on different processors; in such a case, a TFG is constructed from several CFGs of the individual programs. This article adopts a single-program multiple-data programming model to construct our TFG; however, the method could be applied to multiple-program multiple-data programming models with minor revisions. Figure 4 presents a hierarchical BSP program in CFG, where four threads (T_1 to T_4) are divided into four supersteps. In the first superstep, four threads are further grouped into two groups (g_1 to g_2). Each subgroup has its own supersteps.

The notations used are presented in Table I. Given a CFG $G = (V, E)$, comprising a set of nodes V and a set of edges E, we denote the set of immediate successors of a node v by $Succ(v)$ and the set of immediate predecessors of v by $Pred(v)$ (e.g., if there exists an edge $e(v_1, v_2) \in E$, then $Succ(v_1) = v_2$ and $Pred(v_2) = v_1$). For convenience, we denote a set of immediate successors and immediate predecessors of a set V_0 as $Succ(V_0)$ and $Pred(V_0)$, respectively.

$$Succ(V_0) = \bigcup_{v \in V_0} Succ(v)$$

Table I. Notation

G	a CFG $G = (V, E)$, comprising a set of nodes V and a set of edges E
$e(v, u)$	a directed edge from v to u
$w(v, u)$	a walk from v to u, which is a sequence of connected nodes
$W(v, u)$	the set of all walks from v to u
$Succ(v)$	the set of immediate successors of node v
$Pred(v)$	the set of immediate predecessors of node v
$Succ(V)$	the set of immediate successors of a set of node V
$Pred(V)$	the set of immediate predecessors of a set of node V
O	the set of groups in a BSP program
g	a group in a BSP program
$V_S(g)$	the set of begin nodes of group g
$V_E(g)$	the set of end nodes of group g
$O_{SUB}(g)$	the set of immediate subgroups of group g
V_B	a set of barrier nodes
V_{TF}	a set of nodes that belong to a thread fragment
G'	a TFG $G' = (V', E')$, comprising a set of nodes V' and a set of edges E'
$V_T(t)$	the set of nodes with thread t
$T(v)$	the thread that node v belongs to
$\Gamma_{TFG}(V_{TF})$	a mapping function that maps a thread fragment $V_{TF} \subset V$ to a node $v \in G$
$\Gamma_{CFG}(v)$	a mapping function that maps a node $v \in V'$ to a thread fragment $V_{TF} \subset V$

$$Pred(V_0) = \bigcup_{v \in V_0} Pred(v)$$

A *walk* in the graph is a sequence of connected nodes that are not necessarily distinct. We denote a walk from v_0 to v_n in G by $w(v_0, v_n)$

$$w(v_0, v_n) =< v_0, v_1, \ldots, v_{n-1}, v_n >, \forall (1 \leq k \leq n) \wedge (k \in \mathbb{N}) : v_k \in Succ(v_{k-1}),$$

where $\mathbb{N}$ is the set of natural numbers. Let $W(u, v)$ be a set of all walks from u to v in G; note that A $W(u, v)$ wills an infinite set if there is a loop between u and v.

A hierarchical BSP program might have several groups in a superstep. A *group* is a set of threads that are present during a certain period of time; all threads in the set are executed simultaneously and synchronized by barriers belonging to the group. A hierarchical BSP program has an implicit group that contains all threads. Figure 4 contains an implicit group g_0 that contains all threads in the graph. Let O be the set of all groups in a BSP program. A *subgroup* g' of a group g is a set of threads such that $g' \subseteq g$. We say that a group g' is an *immediate subgroup* of a group g if and only if g' is a subgroup of g and there does not exist a group g'' such that g'' is a subgroup of g and g' is a subgroup of g''. The set of immediate subgroups for a group g is denoted by $O_{SUB}(g)$.

Barriers belonging to a group could synchronize threads in the group. Given a group $g \in O$, a set of BSP barrier nodes is denoted by $V_B(g)$, which includes all BSP synchronization nodes in group g. Sets of beginning nodes and ending nodes of a BSP thread are denoted by $V_S(g)$ and $V_E(g)$, respectively, that contain all beginning nodes and all ending nodes of a BSP program in group g.

Barrier nodes block threads, dividing them into thread fragments and forming supersteps. A thread fragment might belong to multiple supersteps, depending on the numbers of prior barriers along a control flow. We say that a barrier that has n prior barriers along a control flow belongs to *generation n*. A barrier node might belong to more than one generation if there are multiple control flows with different numbers of barriers reaching the barrier. For a given group g, let the numbers of BSP barrier

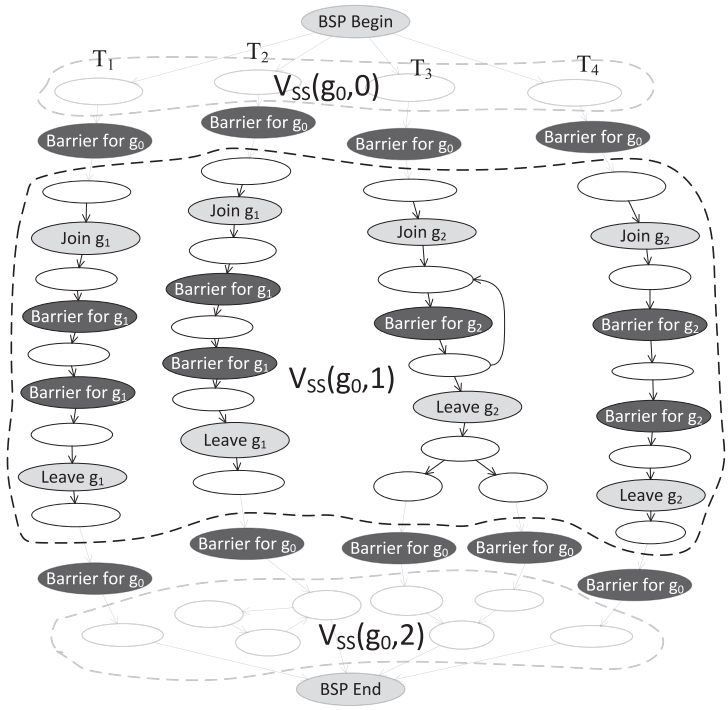

Fig. 5. Supersteps for Figure 4. Nodes inside the dashed area are a superstep, named V_{SS}. Four supersteps are shown in the figure: $V_{SS}(g_0, 0)$, $V_{SS}(g_0, 1)$, $V_{SS}(g_0, 2)$, and $V_{SS}(g_0, 3)$.

nodes in a walk w be $NumBarr(w, g)$. We denote a set of the BSP barriers that have n prior barriers by $V_B(g, n)$, and the sets of the first BSP barriers and last BSP barriers of group g are denoted by $V_B(g, entry)$ and $V_B(g, exit)$, respectively.

$$v_s \in V_S(g), V_B(g, n) = \{v \in V_B \mid \exists w \in W(v_s, v) \wedge NumBarr(w, g) = n\}$$

$$v_s \in V_S(g), V_B(g, entry) = \{v \in V_B \mid \exists w \in W(v_s, v) \wedge NumBarr(w, g) = 0\}$$

$$v_e \in V_E(g), V_B(g, exit) = \{v \in V_B \mid \exists w \in W(v, v_e) \wedge NumBarr(w, g) = 0\}$$

A superstep in a BSP model is formed by nodes between two adjacent barriers of the same group. Figure 5 shows supersteps for Figure 4, where four supersteps are divided by $V_B(g_0)$. For a given group g and two given adjacent barrier generations n and $n+1$, a set of nodes of superstep $V_{SS}(g, n)$ in G could be derived by traversing nodes from successors of $V_B(g, n)$ to $V_B(g, n+1)$.

$$V_{SS}(g, n) = Succ(V_B(g, n)) \cup \{v \in V \mid v \in Succ(V_{SS}(g, n)) \wedge v \notin V_B(g, n+1)\}$$

A walk in a superstep is defined as a sequence of nodes that contains no barriers of the superstep.

$$W_{SS}(u, v, g) = \{w \in W(u, v) \mid \ o \in O_{SUB}(g), NumBarr(w, o) = 0 \wedge NumBarr(w, g) = 0\}$$

A *thread fragment*, denoted by V_{TF}, is a set of neighboring nodes of a superstep that contains no BSP barriers of a group g or its immediate subgroups, which means that a thread fragment of group g in generation n can be characterized into one of the following three cases:

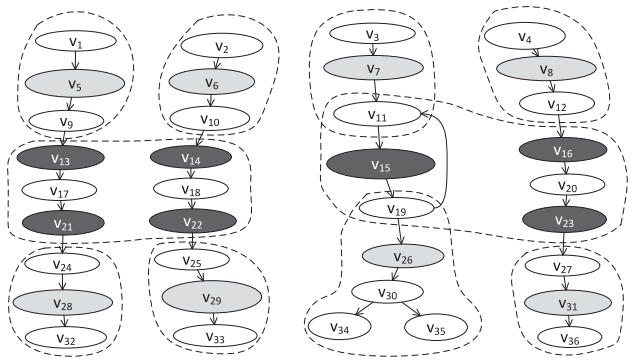

Fig. 6. Superstep $V_{SS}(g_0, 1)$ of Figure 5. Nodes inside a superstep are divided into thread fragments that are used to build a TFG. Nodes v_{11} and v_{19} are overlapped because there is a barrier node v_{15} inside the loop structure. With a barrier node inside a loop, nodes inside the loop may be executed multiple times in different thread fragments; therefore nodes v_{11} and v_{19} appear in different thread fragments, resulting in the thread fragments overlapping.

(a) starts from a successor of barriers $V_B(g, n)$ and ends in a predecessor of barriers in next generation $V_B(g, n+1)$ or barriers in entry barriers of immediate subgroup o of g, $V_B(o, entry)$;
(b) starts from a successor of exit barriers of immediate subgroup o of g, $V_B(o, exit)$, and ends in a predecessor of barriers in next generation $V_B(g, n+1)$ or barriers in entry barriers of another immediate subgroup o' of g, $V_B(o', entry)$; or
(c) starts from the entry barrier of an immediate subgroup o of g, $V_B(o, entry)$, and ends in the exit barrier of o, $V_B(o, exit)$.

For case a, we have a thread fragment $V_{TF}(v, g)$ for each node $v \in Succ(V_B(g, n))$.

$$V_{TF}(v, g) = \{u \mid o' \in O_{SUB}(g), v' \in V_B(g, n+1) \cup V_B(o', entry), \\ W_{SS}(v, u) \neq \emptyset \wedge W_{SS}(u, v') \neq \emptyset\} \quad (1)$$

For case b, we have a thread fragment $V_{TF}(v, o, g)$ for each node $v \in Succ(V_B(o, exit))$.

$$V_{TF}(v, o, g) = \{u \mid o' \in O_{SUB}(g) \wedge o' \neq o, v' \in V_B(o', entry) \cup V_B(g, n+1), \\ W_{SS}(v, u) \neq \emptyset \wedge W_{SS}(u, v') \neq \emptyset\} \quad (2)$$

For case c, we have a thread fragment $V_{TF}(o)$ where o is a subgroup and $o \in O_{SUB}(g)$.

$$V_{TF}(o) = \{v \mid v' \in V_B(o, entry), v'' \in V_B(o, exit), W(v', v) \neq \emptyset \wedge W(v, v'') \neq \emptyset\} \quad (3)$$

Figure 6 shows the thread fragments in superstep $V_{SS}(g_0, 1)$ of Figure 5. Nodes v_{11} and v_{19} are overlapped because there is a barrier node v_{15} inside the loop structure. With a barrier node inside a loop, nodes inside the loop may be executed multiple times in different thread fragments; therefore nodes v_{11} and v_{19} appear in different thread fragments, resulting in the thread fragments overlapping.

We now can construct a TFG for a superstep n of group g in G_{CFG}. A TFG, denoted by $G' = (V', E')$, is a directed graph in which each node is a thread fragment or a grouped thread fragment and each edge is a control flow between nodes. For each V_{TF}, we add a node v to V' to represent V_{TF}. The relation from V_{TF} to the relevant v is denoted by $v = \Gamma_{TFG}(V_{TF})$. Conversely, we denote the relation from $v \in V'$ to V_{TF} by $V_{TF} = \Gamma_{CFG}(v)$.

For two given nodes $v_i \in V'$ and $v_j \in V'$, an edge $e(v_i, v_j)$ is added to E' if and only if there exists an edge $e \in E$ from a node of $\Gamma_{CFG}(v_i)$ to a node of $\Gamma_{CFG}(v_j)$.

$$e(v_i, v_j) \in E' \iff \exists e(v_a, v_b) \in E : v_a \in \Gamma_{CFG}(v_i), v_b \in \Gamma_{CFG}(v_j)$$

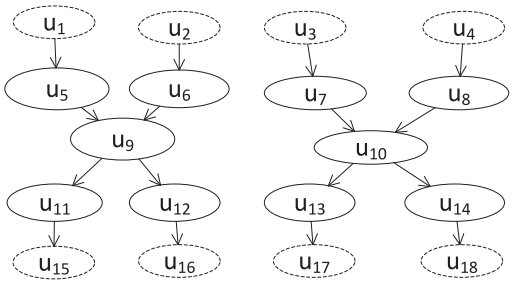

Fig. 7. A TFG for Figure 6.

Nodes of a graph having no predecessor are called *entry nodes*, while those that have no successor are called *exit nodes*. There are multiple entry nodes and exit nodes in a TFG, denoted by $V'(entry)$ and $V'(exit)$, respectively. A thread might have several entry nodes and exit nodes in a TFG. To unify these nodes for each thread, an *initial node* and a *final node* for each thread are introduced into the TFG as an immediate predecessor of all entry nodes of the same thread or an immediate successor of all exit nodes of the same thread, respectively. Given a thread with a thread identification t of a thread, we denote initial nodes and final nodes by $V'(t, initial)$ and $V'(t, final)$. The connections between nodes are indicated as

$$Succ(V'(t, initial)) = \{v \mid v \in V'(entry) \wedge T(v) = t\}$$

and

$$Pred(V'(t, final)) = \{v \mid v \in V'(exit) \wedge T(v) = t\},$$

where $T()$ is a function that returns the thread identification of a thread fragment.

Figure 7 presents a TFG that abstracts the relationship of thread fragments for Figure 6. Nodes u_9 and u_{10} are nodes for thread fragments derived from immediate subgroups with Eq. (3), while the other nodes for thread fragments are derived with Eqs. (1) and (2). Node u_5 is the node for thread fragment $\{v_1, v_5, v_9\}$, that is derived with Eq. (1). Node u_{11} is the node for thread fragment $\{v_{24}, v_{28}, v_{32}\}$ that is derived with Eq. (2). Node u_9 is the node for thread fragment $\{v_{13}, v_{14}, v_{17}, v_{18}, v_{21}, v_{22}\}$ that is derived with Eq. (3). Entry nodes $V'(entry)$ are $\{u_5, u_6, u_7, u_8\}$, exit nodes $V'(exit)$ are $\{u_{11}, u_{12}, u_{13}, u_{14}\}$, the initial nodes are $\{u_1, u_2, u_3, u_4\}$, and the final nodes are $\{u_{15}, u_{16}, u_{17}, u_{18}\}$.

We say that two TFGs G'_0 and G'_1 are *identical* if and only if every node in G'_0 has a node in G'_1 that is related to the same set of fragments, and vice versa.

$$G'_0 \equiv G'_1 \iff \begin{array}{c}(\forall v \in G'_0)(\exists u \in G'_1)(u = \Gamma_{TFG}(\Gamma'_{CFG}(v))) \bigwedge \\ (\forall v \in G'_1)(\exists u \in G'_0)(u = \Gamma'_{TFG}(\Gamma_{CFG}(v)))\end{array}$$

4.2. Constructing TFGs

We designed a TFG construction algorithm that builds the TFG for each BSP superstep from a CFG and performs the lineal thread fragments analysis for each TFG. The idea involves recursively computing concurrency information inside a group. The algorithm ends when any thread encounters an end of a thread or a built TFG is identical to any previously built one.

Algorithm 2 is the kernel algorithm that collects the thread fragment of a designated group as well as constructs the TFG of the group and computes the concurrency information. Algorithm 2 collects thread fragments in case c as mentioned in Section 4.1. The output of the algorithm would be a set of nodes between the entry barrier of an

ALGORITHM 2: TraverseGroup(CFG G, group g, start nodes V_{arg}, blocked nodes V_{blk})

Input: G: The CFG to be analyzed
Input: g: the group to analyze
Input: V_{arg}: a set of starting nodes
Output: V_{blk}: a set of nodes blocked by barriers
Output: V_{gtf}: a set of nodes of CFG, which contains all nodes in the group
Used Data: V_{itr}: a set of nodes of CFG, where nodes are iterators
Used Data: V'_{arg}: a set of starting nodes for a subgroup
Used Data: V'_{blk}: a set of blocked nodes for a subgroup
Used Data: V_{TF}: a set of nodes of CFG, which represents a thread fragment
Used Data: v_a, v_b : nodes of CFG
Used Data: G': a TFG for a superstep
Used Data: v_i, v_j : nodes of TFG
begin
  Initialize V_{itr} with $V_{itr} \leftarrow V_{arg}$
  repeat
    repeat `/* Traverse a superstep of BSP */`
      Let $G' = (V', E')$ be a TFG for the superstep
      foreach $v_a \in V_{itr}$ **do** `/* Traverse thread fragments */`
        $V_{TF} \leftarrow$ `TraverseThreadFragment` (v, V_{TF}, V_{blk})
        Add a node v_i into G' and let $\Gamma_{CFG}(v_i) = V_{TF}$
        Collect traversed nodes by $V_{gtf} \leftarrow V_{gtf} \cup V_{TF}$
      end
      $V_{itr} \leftarrow \emptyset$
      foreach $g' \in O_{SUB}(g)$ **do** `/* Traverse subgroups */`
        Check V_{blk} to determine if every thread belongs to subgroup g' is ready.
        if *g' is ready* **then**
          Let V'_{arg} be the set of nodes encountering barriers of group g'
          Update blocked nodes by $V_{blk} \leftarrow V_{blk} - V'_{arg}$
          Cross barriers before traverse subgroups: $V'_{arg} \leftarrow$ `CrossBarrier` (V'_{arg})
          $V_{TF} \leftarrow$ `TraverseGroup` $(G, g', V'_{arg}, V'_{blk})$
          Add a node v_i into G' and let $\Gamma_{CFG}(v_i) = V_{TF}$
          Collect traversed nodes by $V_{gtf} \leftarrow V_{gtf} \cup V_{TF}$
          Update iterators by $V_{itr} \leftarrow$ `CrossBarrier`(V'_{blk})
        end
      end
    until *V_{itr} is empty*;
    foreach $v_i, v_j \in V'$ **do** `/* Build up edges of G' */`
      if $\exists e(v_a, v_b) \in E$, *where* $v_a \in \Gamma_{CFG}(v_i)$, $v_b \in \Gamma_{CFG}(v_j)$ **then**
        Add $e(v_i, v_j)$ into E'
      end
    end
    Add initial nodes and final nodes, and construct edges for initial and final nodes
    `/* All iterators encountered BSP sync nodes. */`
    `ComputeMHPRegion` (G')
    $V_{itr} \leftarrow$ `CrossBarrier` (V_{blk})
  until $V_{blk} \subseteq V_B(g, exit)$ *or two TFGs are identical*;
  return V_{gtf}
end

immediate subgroup o of g, $V_B(o, entry)$ and the exit barrier of o, $V_B(o, exit)$, which means that Algorithm 2 is an implementation of Eq. (3). Algorithm 3 collects thread fragments of a designated node until barriers, namely cases a and b as mentioned in Section 4.1. Algorithm 3 is an implementation of Eq. (3).

ALGORITHM 3: TraverseThreadFragment(CFG node v, nodes V_{TF}, nodes V_{blk})

```
Input: v: a CFG node to be traversed
Input: V_TF: a set of traversed nodes
Output: V_blk: a set of blocked nodes
Used Data: g': a CFG node
begin
    if v ∈ V_TF then return V_TF                    /* The node is traversed. */
    if v ∈ V_B then                                 /* The node encounter a barrier */
        Add v to V_blk and return V_TF
    end
    Add v into V_TF
    foreach v' ∈ Succ(v) do                         /* Recursively traverse successors */
        V_TF ← V_TF ∪ TraverseThreadFragment (v, V_TF, V_blk)
    end
    return V_TF
end
```

In Algorithm 2, V_{gtf} collects all nodes in the group. Certain temporal variables are introduced to aid the collecting of nodes. V_{arg} is a set of starting nodes in a group, V_{itr} is a set of iterator nodes for recording the locations of iterators, and V_{blk} is a set of blocked nodes that are barrier nodes to identify where iterators are blocked. Algorithm 2 begins by initializing V_{itr} initialized as V_{arg}; nodes in V_{itr} are then traversed recursively based on Algorithm 3. Each node in Algorithm 3 will conform to one of the following conditions:

—be in V_{TF}: the function returns because the iterator is in a circular path;
—be blocked by a barrier: the node is added to V_{blk} and returns to Algorithm 2 because a thread fragment is found; or
—keep traversing to its successors.

Each collected V_{TF} set has a relevant TFG node, and a V_{TF} is added into V_{gtf} for all nodes in a group, which will eventually be a thread fragment of the outer group according to Eq. (3).

Once all of the nodes in V_{itr} have been traversed, we check blocked set V_{blk} because some nodes might be blocked by barriers of subgroups. In such a case we recursively perform TraverseGroup() to traverse a subgroup with designated V'_{arg} and get a thread fragment according to Eq. (3). Two operations are required before invoking TraverseGroup() for subgroups:

—remove nodes of V'_{arg} from V_{blk} because they no longer belong to V_{blk}; and
—since V'_{arg} now contains nodes that are blocked, we have to allow V_{arg} to cross barriers.

CrossBarrier() is a function that helps the input nodes to cross a barrier and outputs a set of nodes after such barriers. After performing TraverseGroup() for each subgroup, the output blocked set V'_{blk} needs to cross barriers; then the processed V'_{blk} set is added to V_{itr} so that thread fragments are collected in the subsequent iterations.

Thread fragments of a superstep are collected when V_{itr} is empty. The next step involves adding edges to complete the TFG. Lineal thread fragments and MHP thread fragments (MTFs) are then analyzed, as explained in detail in Section 4.3.

After processing a superstep, we update V_{itr} with CrossBarrier(V_{blk}) and repeat the procedure to process the next superstep. The algorithm iterates for each superstep until one of two cases is obtained: (1) all blocked nodes V_{blk} are a subset of $V_B(g, exit)$, which means that there are no further supersteps in this group; or (2) two TFGs in a

$$GEN(v) \leftarrow \{v\} \tag{4}$$

$$KILL(v) \leftarrow \emptyset \tag{5}$$

$$IN(v) \leftarrow \bigcup_{v':Pred(v)} OUT(v') \tag{6}$$

$$OUT(v) \leftarrow IN(v) \cup GEN(v) - KILL(v) \tag{7}$$

$$LTF(v) \leftarrow OUT(v) \cup \{v' \mid OUT(v') \ni v\} \tag{8}$$

Fig. 8. Dataflow equations for lineal thread fragments information.

group are identical, which means that there is a loop in the BSP program and we have already explored all possible combinations.

4.3. Lineal Thread Fragments Analysis and MTF

Once the TFG has been constructed, we can compute the concurrent thread fragments of a hierarchical BSP program. Instead of gathering MHP information, we gather nodes that cannot happen in parallel; that is, they have a lineal relation. We collect all nodes along the TFG in our dataflow analysis and maintain the set of entire lineal thread fragments by adding nodes symmetrically so as to keep this set symmetric.

The GEN set obtained by lineal thread fragments analysis is the node itself; the KILL set is always empty; the IN set is the set of ancestor nodes, which is the union of all the predecessor's OUT set; the OUT set is the reached ancestor nodes, which is the union of the IN set and GEN set; and the LTF set is the set of lineal thread fragments derived from the OUT set as follows.

$$LTF(v) \leftarrow OUT(v) \cup \{v' \mid OUT(v') \ni v\}$$

The LTF set gets its ancestor nodes and its children nodes by a symmetric step [Barik 2005]. Figure 8 presents the dataflow equations used to gather lineal thread fragment nodes in a TFG.

After the LTF set of all nodes has been computed, the MTF for each thread fragment is computed as

$$MTF(v) \leftarrow V' - \cup_{v' \in V_T(T(t))} LTF(v'),$$

where $V_T(t)$ denotes a set of nodes with thread t, and $T(v)$ denotes a thread to which node v belongs.

An MHP region is a subgraph of the TFG where thread fragments may be executed concurrently. Nodes belonging to different MHP regions must not be executed in parallel. The MHP regions are determined by first constructing an MHP graph $G'' = (V', E'')$, that is an indirected graph whose nodes are thread fragments and edges are nodes that may happen in parallel; that is, they are related to the *MTF* set.

$$e(v, u) \in E'' \Longleftrightarrow \exists u \in MTF(v) : v \in V'$$

The connected components of a MHP graph, denoted by $\{R_1, \ldots, R_N\}$, are sets of nodes having a walk relationship. We then construct MHP regions $\{S_1, \ldots, S_N\}$ for each connected component. An MHP region, as a subgraph of G', is denoted by $S_n = (R_n, E_n)$, where R_n is a connected component of G'' and where E_n is a set of edges.

$$e(v, u) \in E_n \Longleftrightarrow \exists e(v, u) \in E' : v \in R_n \wedge u \in R_n \tag{9}$$

We compute the lineal thread fragments of nodes and MHP regions for each TFG by running Algorithm 4 with dataflow equations. Table II lists the results of GEN, OUT,

Table II. The GEN, OUT, and LTF Sets for the Example in Figure 7

	GEN	OUT	LTF
u_1	$\{u_1\}$	$\{u_1\}$	$\{u_1, u_5, u_9, u_{11}, u_{12}, u_{15}, u_{16}\}$
u_2	$\{u_2\}$	$\{u_2\}$	$\{u_2, u_6, u_9, u_{11}, u_{12}, u_{15}, u_{16}\}$
u_3	$\{u_3\}$	$\{u_3\}$	$\{u_3, u_7, u_{10}, u_{13}, u_{14}, u_{17}, 18\}$
u_4	$\{u_4\}$	$\{u_4\}$	$\{u_4, u_8, u_{10}, u_{13}, u_{14}, u_{17}, 18\}$
u_5	$\{u_5\}$	$\{u_1, u_5\}$	$\{u_1, u_5, u_9, u_{11}, u_{12}, u_{15}, u_{16}\}$
u_6	$\{u_6\}$	$\{u_2, u_6\}$	$\{u_2, u_6, u_9, u_{11}, u_{12}, u_{15}, u_{16}\}$
u_7	$\{u_7\}$	$\{u_3, u_7\}$	$\{u_3, u_7, u_{10}, u_{13}, u_{14}, u_{17}, 18\}$
u_8	$\{u_8\}$	$\{u_4, u_8\}$	$\{u_4, u_8, u_{10}, u_{13}, u_{14}, u_{17}, 18\}$
u_9	$\{u_9\}$	$\{u_1, u_2, u_5, u_6\}$	$\{u_1, u_2, u_5, u_6, u_9, u_{11}, u_{12}, u_{15}, u_{16}\}$
u_{10}	$\{u_{10}\}$	$\{u_3, u_4, u_7, u_8\}$	$\{u_3, u_4, u_7, u_8, u_{10}, u_{13}, u_{14}, u_{17}, u_{18}\}$
u_{11}	$\{u_{11}\}$	$\{u_1, u_2, u_5, u_6, u_{11}\}$	$\{u_1, u_2, u_5, u_6, u_9, u_{11}, u_{15}\}$
u_{12}	$\{u_{12}\}$	$\{u_1, u_2, u_5, u_6, u_{12}\}$	$\{u_1, u_2, u_5, u_6, u_9, u_{12}, u_{16}\}$
u_{13}	$\{u_{13}\}$	$\{u_3, u_4, u_7, u_8, u_{13}\}$	$\{u_3, u_4, u_7, u_8, u_{10}, u_{13}, u_{17}\}$
u_{14}	$\{u_{14}\}$	$\{u_3, u_4, u_7, u_8, u_{14}\}$	$\{u_3, u_4, u_7, u_8, u_{10}, u_{14}, u_{18}\}$
u_{15}	$\{u_{15}\}$	$\{u_1, u_2, u_5, u_6, u_{11}, u_{15}\}$	$\{u_1, u_2, u_5, u_6, u_9, u_{11}, u_{15}\}$
u_{16}	$\{u_{16}\}$	$\{u_1, u_2, u_5, u_6, u_{12}, u_{16}\}$	$\{u_1, u_2, u_5, u_6, u_9, u_{12}, u_{16}\}$
u_{17}	$\{u_{17}\}$	$\{u_3, u_4, u_7, u_8, u_{13}, u_{17}\}$	$\{u_3, u_4, u_7, u_8, u_{10}, u_{13}, u_{17}\}$
u_{18}	$\{u_{18}\}$	$\{u_3, u_4, u_7, u_8, u_{14}, u_{18}\}$	$\{u_3, u_4, u_7, u_8, u_{10}, u_{14}, u_{18}\}$

Table III. The MTF Set for the Example in Figure 7

	MTF
u_1	$\{u_2, u_3, u_4, u_6, u_7, u_8, u_{10}, u_{13}, u_{14}, u_{17}, u_{18}\}$
u_2	$\{u_1, u_3, u_4, u_5, u_7, u_8, u_{10}, u_{13}, u_{14}, u_{17}, u_{18}\}$
u_3	$\{u_1, u_2, u_4, u_5, u_6, u_8, u_9, u_{11}, u_{12}, u_{15}, u_{16}\}$
u_4	$\{u_1, u_2, u_3, u_5, u_7, u_8, u_9, u_{11}, u_{12}, u_{15}, u_{16}\}$
u_5	$\{u_2, u_3, u_4, u_6, u_7, u_8, u_{10}, u_{13}, u_{14}, u_{17}, u_{18}\}$
u_6	$\{u_1, u_5, u_3, u_4, u_7, u_8, u_{10}, u_{13}, u_{14}, u_{17}, u_{18}\}$
u_7	$\{u_1, u_2, u_4, u_5, u_6, u_8, u_9, u_{11}, u_{12}, u_{15}, u_{16}\}$
u_8	$\{u_1, u_2, u_3, u_5, u_7, u_8, u_9, u_{11}, u_{12}, u_{15}, u_{16}\}$
u_9	$\{u_3, u_4, u_7, u_8, u_{10}, u_{13}, u_{14}, u_{17}, u_{18}\}$
u_{10}	$\{u_1, u_2, u_5, u_6, u_9, u_{11}, u_{12}, u_{15}, u_{16}\}$
u_{11}	$\{u_3, u_4, u_7, u_8, u_{10}, u_{12}, u_{13}, u_{14}, u_{16}, u_{17}, u_{18}\}$
u_{12}	$\{u_3, u_4, u_7, u_8, u_{10}, u_{11}, u_{13}, u_{14}, u_{15}, u_{17}, u_{18}\}$
u_{13}	$\{u_1, u_2, u_5, u_6, u_9, u_{11}, u_{12}, u_{14}, u_{15}, u_{16}, u_{18}\}$
u_{14}	$\{u_1, u_2, u_5, u_6, u_9, u_{11}, u_{12}, u_{13}, u_{15}, u_{16}, u_{17}\}$
u_{15}	$\{u_3, u_4, u_7, u_8, u_{10}, u_{12}, u_{13}, u_{14}, u_{16}, u_{17}, u_{18}\}$
u_{16}	$\{u_3, u_4, u_7, u_8, u_{10}, u_{11}, u_{13}, u_{14}, u_{15}, u_{17}, u_{18}\}$
u_{17}	$\{u_1, u_2, u_5, u_6, u_9, u_{11}, u_{12}, u_{14}, u_{15}, u_{16}, u_{18}\}$
u_{18}	$\{u_1, u_2, u_5, u_6, u_9, u_{11}, u_{12}, u_{13}, u_{15}, u_{16}, u_{17}\}$

and LTF sets for the example in Figure 5. As listed in Table II, the GEN set contains the node itself; the OUT set is the collection of GEN sets along the TFG; and the LTF set is derived from the OUT set by a symmetric step. The MTF set of the example is listed in Table III, which is derived from LTF set by subtraction.

5. MULTITHREAD POWER-GATING ANALYSIS

The TFCA results and the component usage for a power-gating candidate C_i of all concurrent thread fragments can be categorized in the following three cases.

ALGORITHM 4: ComputeMHPRegion(TFG G'): Computing the lineal thread fragments of nodes and the MHP region for TFG G'

Input: G': a TFG $G' = (V', E')$
Output: The MHP regions
Used Data: v, v': TFG nodes
begin
  foreach $v \in V'$ **do** /* Initialize GEN() sets */
    $GEN(v) \leftarrow \{v\}$
  end
  repeat /* Perform data-flow analysis */
    foreach $v \in V'$ **do**
      $IN(v) \leftarrow \bigcup_{v' \in Pred(v)} OUT(v')$
      $OUT(v) \leftarrow IN(v) \cup GEN(v)$
      foreach $v' \in OUT(v)$ **do** /* Compute lineal thread fragments */
        Perform symmetric step: $LTF(v') \leftarrow LTF(v') \cup OUT(v') \cup \{v\}$
      end
    end
  until *for all* $v \in V'$, $IN(v)$ *and* $OUT(v)$ *are converge*;
  foreach $v \in V'$ **do** /* Compute the MHP thread fragments */
    $MTF(v) \leftarrow V' - \cup_{v' \in V_T(T(t))} LTF(v')$
  end
  foreach $v \in V'$ **do** /* Construct the MHP graph */
    foreach $u \in MTF(v)$ **do**
      Add edge $e(v, u)$ to E''
    end
  end
  Find all connected components $\{R_1, \ldots, R_N\}$ of MHP graph G''
  Construct MHP regions $\{M_1, \ldots, M_N\}$ of G' from $\{R_1, \ldots, R_N\}$
end

—*C_i is used by only one thread fragment*. In this case the Sink-N-Hoist framework should be applied to the thread fragment to insert traditional power-gating instructions because the uncertainty of component usage in a multithread program is not present.
—*C_i is not used*. In this case we do not need to handle C_i because a power-gating candidate is defined to be turned off at the beginning of a superstep. This is the optimal case to use because there will be no extra cost and the power savings will be maximal.
—*C_i is used by multiple thread fragments*. When the analysis results indicate that multiple thread fragments might use C_i in an MHP region, we have two strategies for placing PPG instructions. The evaluation is described in detail next.

In this section we present an MTPGA scheme based on the PPG mechanism and TFCA scheme results to estimate energy consumption and to insert power-gating instructions. MTPGA generally inserts a pair of predicated power-on and predicated power-off operations at the positions where a power-gating candidate is first activated and last deactivated for each thread within a MHP region according to the proposed cost model. Figure 9 illustrates a simple scenario in which thread fragments TF_1 and TF_2 may happen in parallel—and thus TF_1 and TF_2 form an MHP region—and CADFA exposes the utilization status of three power-gating candidates, labeled as C_1, C_2, and C_3; in-use units are depicted with light gray boxes in the figure.

Figure 10 demonstrates two possible placements of PPG operations based on the MTPGA results. In most cases, MTPGA will place a pair of PPG operations for each

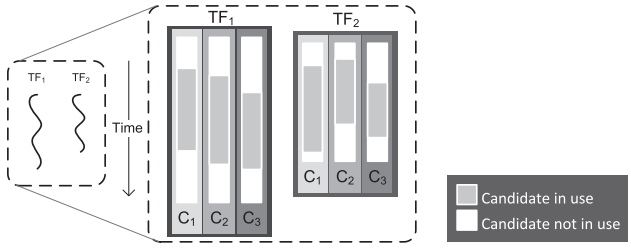

Fig. 9. Two thread fragments TF_1 and TF_2 in an MHP region and their utilization status for the power-gating candidates C_1, C_2, and C_3; in-use units are depicted with light gray boxes.

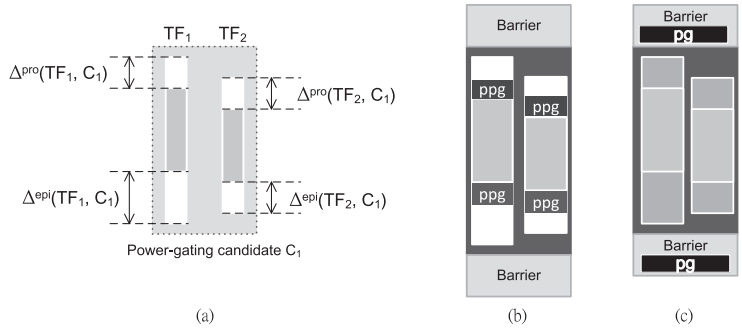

Fig. 10. Two kinds of instruction placement for power gating: (a) Two concurrent thread fragments TF_1 and TF_2 using a power-gating candidate C_1 and their component usage; (b); (c) two strategies for placing power-gating instructions among TF_1 and TF_2; (b) the leakage energy of thread fragments is worthy of being gated (i.e., the calculation result of Eq. 10); thus all thread fragments would be inserted with ppg instructions; (c) the leakage energy of thread fragments is not worth gating; in such a case, we insert conventional power-gating instructions before and after the MHP region.

power gating candidate if appropriate. With the support of conditional execution of power-gating instructions, power gating will only occur when a unit is first activated and last deactivated within an MHP region. Recall that the proposed PPG mechanism incorporates a set of reference counters (N reference counters for N power-gating candidates) for tracking the number of threads that have been referenced, and predicated bits are set only when the value of their corresponding reference counters is 0. Therefore, within the MHP region there will be only a pair of power-gating operations, namely the first power-on and the last power-off operations, belonging to a pair of PPG operations being executed, whereas the power gating of the other PPG operations will be disabled. This ensures that a unit will be alive whenever it is required for processing.

PPG is not cost free, and we now take this into consideration when building a model for determining which PPG placement strategy should be employed. The model is based on the comparison of the energy cost between normal power gating and the PPG in an MHP region. Suppose that there are N power-gating candidate units, C_1, $C_2, \ldots, C_N$, and K thread fragments in an MHP region, TF_1, $TF_2, \ldots, TF_K$. We define two functions, named Δ^{pro} and Δ^{epi}, that take a thread and a power-gating candidate as their parameters for computing the inactive period of the power-gating candidate before/after the candidate operates for the first/last time within the thread as

$$\Delta^{pro}(TF_i, C_j) = start(TF_i, C_j) - start(TF_i)$$

and

$$\Delta^{epi}(TF_i, C_j) = end(TF_i) - end(TF_i, C_j),$$

where $start(TF_i, C_j)$ returns the time that C_j is first used in TF_i and $start(TF_i)$ returns the start time of TF_i, while $end(TF_i, C_j)$ returns the time that C_j is last used in TF_i and $end(TF_i)$ returns the end time of TF_i. Figure 10 portrays the implications of the aforesaid functions with TF_1 and C_1 as parameters. Furthermore, we define that $\Delta^{pro}_{min}(C_j)$ and $\Delta^{epi}_{min}(C_j)$ return the minimum of $\Delta^{pro}(TF_i, C_j)$ and $\Delta^{epi}(TF_i, C_j)$ in terms of all T_i as

$$\Delta^{pro}_{min}(C_j) = \mathbb{MIN}_{\forall i \in K} \Delta^{pro}(TF_i, C_j)$$

and

$$\Delta^{epi}_{min}(C_j) = \mathbb{MIN}_{\forall i \in K} \Delta^{epi}(TF_i, C_j),$$

where $\Delta^{pro}_{min}(C_j)$ represents the earliest time that C_j might be used after the MHP region starts, $\Delta^{epi}_{min}(C_j)$ represents the latest time that C_j might be used prior to the MHP region ending, and $\mathbb{MIN}$ is a function that returns the minimum value of its parameters. Accordingly, the energy consumption $\mathbb{E}_{pred}$ of the PPG control within the MHP region is

$$\begin{aligned}\mathbb{E}_{pred}(TF, C_j) = {} & \mathbb{E}_{on}(C_j) + \mathbb{E}_{off}(C_j) + K_j \times (\mathbb{E}_{p-on} + \mathbb{E}_{p-off}) \\ & + (\Delta^{pro}_{min}(C_j) + \Delta^{epi}_{min}(C_j)) \times \mathbb{P}_{rleak}(C_j),\end{aligned} \tag{10}$$

where functions $\mathbb{E}_{on}(C_j)$ and $\mathbb{E}_{off}(C_j)$ return the energy consumption of issuing power-on and power-off instructions for component C_j, respectively; K_j represents the number of threads in the MHP region that requires C_j to operate; $\mathbb{E}_{p-on}$ and $\mathbb{E}_{p-off}$ are the energy consumptions associated with operating a set of predicated power-on and predicated power-off manipulation operations described in Section 3.1, excluding the power-on and power-off operations, respectively; and $\mathbb{P}_{rleak}(C_j)$ represents the leakage-power consumption of C_j in a cycle when the power supply is gated. In contrast, when we employ normal power-gating control at the beginning and end of the MHP region rather than applying the PPG management, the energy consumption $\mathbb{E}_{normal}$ of such operations and the potential leakage dissipation is

$$\mathbb{E}_{normal}(TF, C_j) = \mathbb{E}_{on}(C_j) + \mathbb{E}_{off}(C_j) + (\Delta^{pro}_{min}(C_j) + \Delta^{epi}_{min}(C_j)) \times \mathbb{P}_{leak}(C_j), \tag{11}$$

where $\mathbb{P}_{leak}(C_j)$ represents the leakage power consumption of unit C_j during a cycle.

Accordingly, we can derive the following inequality for ensuring the worthiness of PPG

$$\mathbb{E}_{pred}(TF, C_j) < \mathbb{E}_{normal}(TF, C_j),$$

and substituting $\mathbb{E}_{pred}(TF, C_j)$ and $\mathbb{E}_{normal}(TF, C_j)$ into Eqs. (10) and (11), respectively, yields

$$\begin{aligned}&\mathbb{E}_{on}(C_j) + \mathbb{E}_{off}(C_j) + K_j \times (\mathbb{E}_{pred-on} + \mathbb{E}_{pred-off}) + (\Delta^{pro}_{min}(C_j) + \Delta^{epi}_{min}(C_j)) \times \mathbb{P}_{rleak}(C_j) \\ &\quad < \mathbb{E}_{on}(C_j) + \mathbb{E}_{off}(C_j) + (\Delta^{pro}_{min}(C_j) + \Delta^{epi}_{min}(C_j)) \times \mathbb{P}_{leak}(C_j).\end{aligned}$$

Thus we have

$$\Delta^{pro}_{min}(C_j) + \Delta^{epi}_{min}(C_j) > \frac{K_j \times (\mathbb{E}_{pred-on} + \mathbb{E}_{pred-off})}{\mathbb{P}_{leak}(C_j) - \mathbb{P}_{rleak}(C_j)}$$

as the criterion for determining whether a PPG should be applied. Algorithm 5 is an implementation of the proposed MTPGA. The algorithm receives an MHP region as its input and decides which power-gating policy to adopt. Basically, it determines whether a PPG should be employed for each MHP region.

ALGORITHM 5: Algorithm of MTPGA.

Input : A multithread program and its MHP and CADFA with Sink-N-Hoist information.
Output: The program with power-gating controls.

foreach *MHP region* **do**
  foreach *power-gating candidate C* **do**
    Find entry nodes and exit nodes of the MHP region and compute $\Delta_{min}^{pro}(C)$ and $\Delta_{min}^{epi}(C)$
    if $\Delta_{min}^{pro}(C) + \Delta_{min}^{epi}(C) \leq THRESHOLD^{\dagger}$ **then**
      Place a power-on and a power-off instruction for C at the beginning and the end of the MHP region, respectively.
    else
      foreach *thread* **do**
        Place a predicated-power-on and a predicated-power-off operation before/after the candidate operates for the first/last time within the thread.
      end
    end
  end
end

$^{\dagger}THRESHOLD = \dfrac{K_j \times (\mathbb{E}_{pred-on} + \mathbb{E}_{pred-off})}{\mathbb{P}_{leak}(C) - \mathbb{P}_{rleak}(C)}$

6. EXPERIMENT AND DISCUSSION

6.1. Platform

We used a DEC-Alpha-compatible architecture with the PPG controls and two-way to 8-way simultaneous multithreading as the target architecture for our experiments. The equipped SMT machine replicated certain resources for each thread such as program counter and registers, while function units were shared with both threads. The proposed MTPG framework was evaluated by a post-estimated SMT simulator based on Wattch toolkits with a 0.10μm process parameter and a 1.9-V supply voltage. The SMT simulator schedules multiple Wattch simulators to execute programs separately, and then reschedules the component usage on a cycle-by-cycle basis (according to execution traces gathered from Wattch simulators) to estimate the execution time and power consumption.

Table IV summarizes the baseline configuration of the Wattch simulators in our experiments. By default, the simulator performs out-of-order execution. We used the "-issue:inorder" option in the configuration so that instructions would be executed in order, which ensures the correctness of execution. Nevertheless, our approach could also be applied to machines issuing out-of-order execution commands when additional hardware supports are employed such as in the hardware proposed in You et al. [2006].

Figure 11 illustrates the phases of the compilation. Two phases were added in order to analyze the component usage of a BSP program: the concurrent thread fragments phase (see Section 4) and the low-power optimization phase (see Section 5). The TFCA phase is performed in low SUIF, which constructs the TFG and analyzes the concurrency among threads. We incorporate the low-power optimization phase just before code generation, that is, after all traditional performance optimizations were performed. Hence, the additional phase hardly influences the performance; it only inserts power-gating instructions or PPG instructions and thus barely affects the execution behavior. The implementation was based on SUIF2 and the *CFG* and *machine* libraries from Machine-SUIF. Programs were first transformed from high SUIF into low SUIF

Table IV. Baseline Processor Configuration

Parameter	Configuration	Parameter	Configuration
Clock	600 MHz	Function units	4 integer ALU
Processor parameters	0.10 μm,1.9V		1 integer mul/div unit
Issue	In-order		4 floating-point ALU
Decode width	8 instruction/cycle		1 floating-point mul/div unit
Issue width	8 instruction/cycle	Register File	32 64-bit integer registers
Commit width	8 instruction/cycle		32 64-bit floating-point registers
RUU size	128		1 power-gating control register
LSQ size	64		

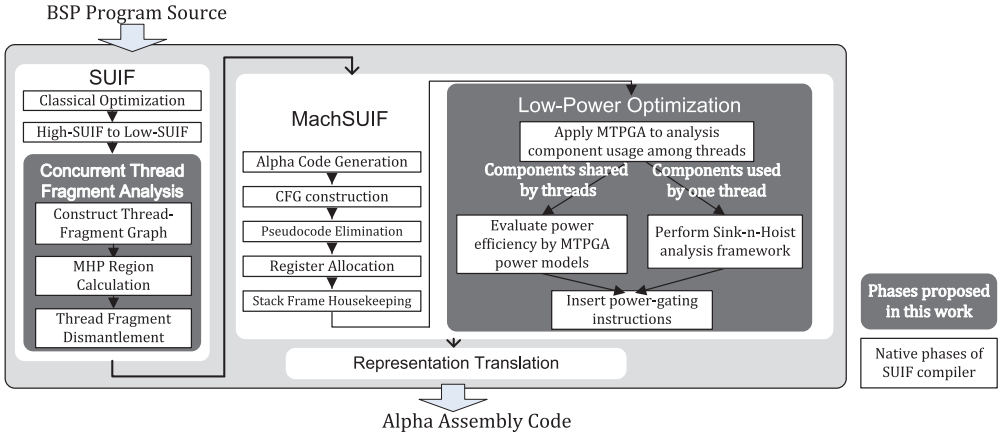

Fig. 11. Power management in the compilation phases of multithread programs.

format with SUIF, processed by concurrent thread fragment analysis, and then translated to the machine- or instruction-level CFG form with Machine-SUIF. Four components of the low-power optimization phase for multithread programs (implemented as a Machine-SUIF pass) were then performed, and finally, the compiler generated DEC Alpha assembly code with extended power-gating controls.

The power-gating mechanism is absent in the original DEC Alpha processor, hence there are no power-gating instructions in its instruction set. Moreover, programs could be roughly categorized into compiled user source codes and libraries, and there are no directives in executables for distinguishing one from another; however, the absence of source codes prevents power-gating analysis. We therefore defined a set of special instructions as power-gating instructions so that they could be recognized by the DEC Alpha assembler and linker: "stl $24, negative_offset($31)", where the *negative offset* is a negative integer used for indicating the function units to be powered on or off and the boundary for kernel extraction. The $31 register in the DEC Alpha processor is a constant zero register, and so the instruction stores a value at a negative address that is invalid and should not be generated by a standard compiler. We made a small modification in Wattch to prevent the processor from accessing such invalid memory addresses: when the instruction decoder deciphered such instructions, it extracted the user directive information and converted it to an NOP (no-operation) instruction. Furthermore, since Wattch does not model leakage at the component level per se, we assumed that leakage power contributes 10% or 30% of the total power consumption [Butts and Sohi 2000; Rusu et al. 2007]. We also assumed that wakeup operations of

Table V. Parameter Settings Used for Generating TFGs

Parameter	Setting A	Setting B	Setting C
Number of nodes	6-10	11-16	11-20
Out degree	2	2	4
In degree	2	2	4
Number of layers	5	8	5
Size of layer	2	2	4
Number of samples	3592	4971	2283

power-gating controls have an eight-cycle latency and that it took 14× the leakage energy per cycle to power a component off and on [Hu et al. 2004]. It was further assumed that the energy consumption associated with fetching and decoding a power-gating instruction was twice the leakage power. The overhead energy of the additional predicated power-gating controller (PPGC) was also considered. According to the synthesis result of PPGC by Synopsys Design Compiler, we assumed that the PPGC took 4×10^{-4} times the power of integer ALU.

We report the power usage of analyzed code regions (i.e., source codes from the user), not including the power usage that is not associated with the user program (e.g., libraries and the C runtime system). Also, the baseline data was provided by the power estimation of Wattch *cc3* with a clock-gating mechanism that gates the clocks of unused resources in multiport hardware to reduce the dynamic power; however, leakage power still exists.

6.2. Simulation Results

To verify our proposed MTPGA algorithm and PPG mechanism, we focused on investigating component utilization in the supersteps. We report two sets of simulation results: one for random TFGs and the other for BSP programs converted from OpenCL kernels. Each set of results compares three types of experiments: (1) no power-gating mechanism (baseline), (2) CADFA with a conventional power-gating mechanism from a previous work [You et al. 2002, 2006], and (3) MTPG with the PPG mechanism.

We first generated random TFGs and applied small programs as thread fragments. Random TFGs were generated using GGen, which is a random graph generator for scheduling simulations [Cordeiro et al. 2010]. The generation method was a slightly modified version of fanin/fanout method. We added a parameter for the size of layer to control the shape of generated graphs, where a layer is a set of nodes without edges. We generated random edges between adjacent layers only, which forced the generated graphs to fit the D-BSP communication rule. Also, a label swapping phase was added immediately before generating the graph to increase the randomness of thread fragments. Each node in the generated TFGs was mapped to a floating-point DSPstone [Zivojnovic et al. 1994] program. The random TFGs were all DAGs and generated with parameters as follows:

—*number of nodes*: the number of thread fragments in the graph;
—*out-degree*: the out-degree of each node controls the number of success of a thread fragment;
—*in-degree*: the in-degree of each node controls the number of predecessors of a thread fragment;
—*number of layers*: the number of layers in the graph;
—*size of layer*: the number of nodes in a layer controls the size of hardware threads.

The energy consumption results for the parameter settings in Table V are listed in Tables VI, VIII, IX, and X. We used 10,756 graph instances to evaluate all settings. All

Table VI. Normalized Total Energy Consumptions of Randomly Generated TFGs for Setting A on Leakage Contribution Set to 10% and 30% (see Table V), Categorized by the Number of MHP Regions for Cases with Two Hardware Threads

Leakage contribution set to 10%						
number of MHP regions	method	dynamic	leakage[a]	leakage[b]	overhead	total
1	baseline	52.55%	12.35%	35.10%	0.00%	100.00%
	CADFA	52.66%	2.74%	36.91%	9.14%	101.46%
	MTPG	52.57%	9.96%	35.26%	0.45%	98.24%
2	baseline	50.60%	12.86%	36.54%	0.00%	100.00%
	CADFA	50.70%	2.73%	38.31%	7.63%	99.36%
	MTPG	50.63%	7.72%	36.90%	0.89%	96.14%
3	baseline	49.80%	13.07%	37.13%	0.00%	100.00%
	CADFA	49.89%	2.74%	38.90%	7.20%	98.72%
	MTPG	49.83%	6.03%	37.76%	1.33%	94.96%
4	baseline	48.70%	13.35%	37.94%	0.00%	100.00%
	CADFA	48.80%	2.73%	39.85%	6.40%	97.77%
	MTPG	48.75%	4.82%	38.81%	1.71%	94.09%
5	baseline	47.64%	13.63%	38.73%	0.00%	100.00%
	CADFA	47.73%	2.70%	40.71%	5.68%	96.83%
	MTPG	47.70%	4.08%	40.01%	2.11%	93.90%
Leakage contribution set to 30%						
number of MHP regions	method	dynamic	leakage[a]	leakage[b]	overhead	total
1	baseline	22.52%	20.15%	57.33%	0.00%	100.00%
	CADFA	22.56%	4.50%	60.29%	15.28%	102.63%
	MTPG	22.53%	16.26%	57.60%	0.73%	97.12%
2	baseline	21.11%	20.52%	58.37%	0.00%	100.00%
	CADFA	21.15%	4.36%	61.18%	12.39%	99.08%
	MTPG	21.12%	12.35%	58.95%	1.43%	93.84%
3	baseline	20.56%	20.66%	58.78%	0.00%	100.00%
	CADFA	20.60%	4.33%	61.56%	11.55%	98.05%
	MTPG	20.58%	9.55%	59.77%	2.09%	91.99%
4	baseline	19.85%	20.84%	59.30%	0.00%	100.00%
	CADFA	19.89%	4.26%	62.26%	10.13%	96.55%
	MTPG	19.87%	7.56%	60.64%	2.67%	90.74%
5	baseline	19.22%	21.01%	59.77%	0.00%	100.00%
	CADFA	19.26%	4.18%	62.78%	8.89%	95.11%
	MTPG	19.25%	6.34%	61.68%	3.24%	90.52%

[a]leakage energy consumed by power-gateable units.
[b]leakage energy consumed by other units.

results are normalized to the situation without a power-gating mechanism. The total energy consumption is divided into four categories: (1) the dynamic energy dissipated by the processor, (2) the leakage energy dissipated by power-gateable units, (3) the leakage energy dissipated by the entire processor except for power-gateable units, and (4) the overhead due to extra power-gating instructions. The overhead includes the energy consumed by power-gating instructions, the energy consumed due to the latency caused by powering on components that have been incorrectly powered off, and the energy consumed by the predicated power-gating controller. Settings A and B are for machines equipped with two hardware threads, while Setting C is for those equipped with four hardware threads. With MTPG, the total power consumption for each setting was reduced to 93.90%, 93.32%, and 95.12%, respectively, relative to the baseline (i.e., no

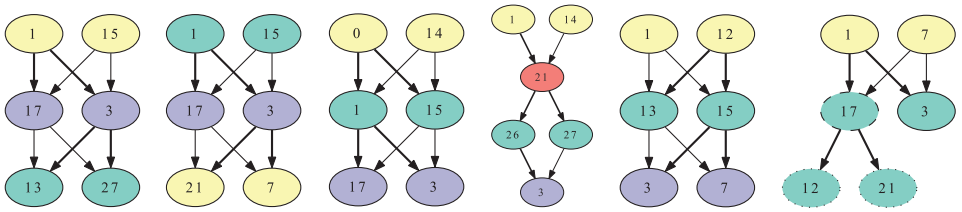

Fig. 12. Selected best cases of randomly generated TFGs.

Table VII. Mapping from Node Labels to Benchmark Programs

label	benchmarks	label	benchmarks
0, 14	complex multiply	7, 21	iir biquard one section
1, 15	complex update	8, 22	lms
2, 16	convolution	9, 23	mat1x3
3, 17	dot product	10, 24	matrix1
4, 18	fir2dim	11, 25	n complex updates
5, 19	fir	12, 26	n real updates
6, 20	iir biquard n section	13, 27	real update

power-gating mechanism) on leakage contribution set to 10%. On leakage contribution set to 30% with MTPG, the total power consumption for each setting was reduced to 90.52%, 89.51%, and 91.94%, respectively, relative to the baseline.

Figure 12 demonstrates six TFGs representing the best cases for Setting A. As mentioned previously, the graph nodes in the figure are thread fragments and graph edges are dependencies between thread fragments. The color of a node represents the MHP region the node belongs to, which is computed via Algorithm 4; nodes with the same color belong to the same MHP region. There are three MHP regions in Figures 12(a), 12(b), 12(c), and 12(e), four MHP regions in Figure 12(d), and two MHP regions in Figure 12(f). The line styles of the node borders represent their types in an MHP region: dashed lines indicate entry thread fragments, dotted lines indicate exit thread fragments, and solid lines indicate both entry and exit thread fragments. The type of nodes is used to evaluate the insertion of power-gating instructions using Algorithm 5. Each node is labeled with a unique number that maps to a program. Table VII lists the mapping from the node labels to DSPstone programs. Note that each DSPstone program is mapped to labels twice for samples with two hardware threads in order to generate random graphs covering both heterogeneous and homogeneous cases; for samples with four hardware threads, each DSPstone program is mapped to four labels for the same reason. MTPG reduced the total energy consumption by an average of about 11% in the cases shown in Figure 12 on leakage contribution set to 30%, namely to 87.79%, 88.15%, 88.27%, 88.39%, 88.59%, and 88.78% relative to the baseline for the cases shown in Figure 12(a) to 12(f), respectively.

Table VI lists the energy consumption results for Setting A with different numbers of MHP regions: on leakage power contribution set to 10% with MTPG, the total energy consumption was 98.24%, 96.14%, 94.96%, 94.09%, and 93.90% for one to five MHP regions, respectively; while on leakage power contribution set to 30% with MTPG, the total energy consumption was 97.12%, 93.84%, 91.99%, 90.74%, and 90.52% for one to five MHP regions, respectively. The results indicate that the energy consumption by the random sample reduced as the number of MHP regions increased, with the trend stabilizing for more than four MHP regions. As indicated in Table VI, while CADFA results in less leakage energy in power-gateable units (about 30% energy consumption

Table VIII. Normalized Total Energy Consumptions of Randomly Generated TFGs for Setting B

Leakage contribution set to 10%						
number of MHP regions	method	dynamic	leakage[a]	leakage[b]	overhead	total
1	baseline	54.79%	11.77%	33.44%	0.00%	100.00%
	CADFA	54.89%	2.70%	35.13%	10.30%	103.02%
	MTPG	54.80%	10.54%	33.53%	0.26%	99.13%
2	baseline	51.23%	12.69%	36.07%	0.00%	100.00%
	CADFA	51.33%	2.68%	37.71%	7.62%	99.34%
	MTPG	51.25%	9.67%	36.25%	0.56%	97.73%
3	baseline	51.26%	12.69%	36.05%	0.00%	100.00%
	CADFA	51.36%	2.68%	37.67%	7.74%	99.45%
	MTPG	51.29%	8.42%	36.32%	0.76%	96.79%
4	baseline	51.20%	12.70%	36.10%	0.00%	100.00%
	CADFA	51.29%	2.69%	37.65%	7.53%	99.16%
	MTPG	51.23%	6.86%	36.45%	1.01%	95.55%
5	baseline	50.64%	12.85%	36.51%	0.00%	100.00%
	CADFA	50.73%	2.69%	38.20%	7.29%	98.91%
	MTPG	50.67%	6.06%	37.04%	1.26%	95.04%
6	baseline	50.45%	12.90%	36.65%	0.00%	100.00%
	CADFA	50.54%	2.64%	38.24%	6.99%	98.41%
	MTPG	50.49%	5.46%	37.26%	1.47%	94.69%
7	baseline	48.05%	13.52%	38.43%	0.00%	100.00%
	CADFA	48.15%	2.67%	40.13%	5.56%	96.50%
	MTPG	48.10%	4.77%	39.13%	1.65%	93.65%
8	baseline	48.75%	13.34%	37.91%	0.00%	100.00%
	CADFA	48.83%	2.72%	39.37%	5.29%	96.21%
	MTPG	48.79%	4.50%	38.60%	1.42%	93.32%

[a]leakage energy consumed by power-gateable units.
[b]leakage energy consumed by other units.

relative to MTPG), it suffers the overhead of traditional power-gating instructions (about 11× the energy consumption relative to MTPG). This overhead is mostly due to the additional cycles required to internally turn on components that are incorrectly turned off. The leakage energy consumed by ones other than power-gateable ones increases in both CADFA and MTPG because of the extra power-gating instructions in that extra power-gating instructions affect instruction fetching, which results in more execution cycles when power-gating instructions are not present and thus increase the leakage energy.

Tables VIII and IX list energy consumption results for Setting B categorized by the number of MHP regions. Compared to Setting A, Setting B generates TFGs with more thread fragments and more layers. The best energy-saving result for Setting B with MTPG was 89.51% energy consumption relative to no power-gating mechanism when there are eight MHP regions on leakage contribution set to 30%. Similar to the experimental results for Setting A, the trend stabilizes when there are more than five MHP regions. Table X lists the energy consumption results for Setting C categorized by the number of MHP regions. Setting C generates TFGs for hardware equipped with four hardware threads. The best energy-saving result for Setting C with MTPG was 91.94% energy consumption relative to no power-gating mechanism when there are five MHP regions on leakage contribution set to 30%. The CADFA results indicates how a large amount of overhead energy could be consumed by incorrectly inserted power-gating instructions. With the traditional CADFA method, the samples in one MHP region consumed 117.43% energy relative to no power-gating mechanism on leakage

Table IX. Normalized Total Energy Consumptions of Randomly Generated TFGs for Setting B

Leakage contribution set to 30%						
number of MHP regions	method	dynamic	leakage[a]	leakage[b]	overhead	total
1	baseline	24.06%	19.75%	56.19%	0.00%	100.00%
	CADFA	24.11%	4.56%	59.03%	17.57%	105.26%
	MTPG	24.07%	17.69%	56.34%	0.44%	98.54%
2	baseline	21.48%	20.42%	58.10%	0.00%	100.00%
	CADFA	21.52%	4.32%	60.73%	12.37%	98.95%
	MTPG	21.49%	15.57%	58.38%	0.89%	96.33%
3	baseline	21.52%	20.41%	58.07%	0.00%	100.00%
	CADFA	21.56%	4.32%	60.67%	12.60%	99.14%
	MTPG	21.53%	13.55%	58.50%	1.23%	94.81%
4	baseline	21.45%	20.43%	58.12%	0.00%	100.00%
	CADFA	21.49%	4.32%	60.62%	12.21%	98.65%
	MTPG	21.46%	11.04%	58.69%	1.62%	92.81%
5	baseline	21.09%	20.52%	58.39%	0.00%	100.00%
	CADFA	21.13%	4.30%	61.07%	11.76%	98.26%
	MTPG	21.10%	9.69%	59.23%	2.02%	92.05%
6	baseline	20.93%	20.56%	58.51%	0.00%	100.00%
	CADFA	20.97%	4.21%	61.05%	11.22%	97.45%
	MTPG	20.95%	8.71%	59.49%	2.35%	91.49%
7	baseline	19.41%	20.96%	59.63%	0.00%	100.00%
	CADFA	19.45%	4.14%	62.23%	8.64%	94.46%
	MTPG	19.43%	7.40%	60.71%	2.56%	90.09%
8	baseline	19.79%	20.86%	59.35%	0.00%	100.00%
	CADFA	19.82%	4.25%	61.64%	8.28%	93.99%
	MTPG	19.80%	7.04%	60.45%	2.22%	89.51%

[a]leakage energy consumed by power-gateable units.
[b]leakage energy consumed by other units.

contribution set to 30%. These results reveal that our method is practical for both hardware configurations.

Focus is now directed to examining our method using BSP benchmarks. We used three BSP programs from BSPedupack, a library of numerical algorithms written in C according to the BSP model [Bisseling 2004]. Four programs of BSPedupack were applied to examine our optimization method, including *fft*, *inprod*, *lu*, and *matvec*. Figure 13 shows the the energy consumption normalized to the baseline case with no power-gating mechanism. On leakage contribution set to 10%, the average reduction in total energy consumption was 4.32%, and was largest for *mv* (7.52%) and the smallest for *lu* (2.27%). On leakage contribution set to 30%, the average reduction in total energy consumption was 8.32%, and was largest for *mv* (13.23%) and the smallest for *lu* (5.30%).

We then evaluated our method using BSP programs from OpenCL-based kernels. OpenCL is an industry attempt to provide standards for GPGPU and heterogeneous multicore programming. An OpenCL program can be roughly divided into host code and kernel code, where the host code is executed on an MPU and the kernel code on OpenCL devices. The OpenCL kernel codes comprise concurrent threads with global barriers, making it easy to transfer them into BSP programs. We incorporated kernel serialization to avoid the threading overhead in parallel kernel execution and to handle synchronization for barriers in kernel functions. We adopt a work-item coalescing scheme [Lee et al. 2010] for kernel serialization, which serializes kernel execution by enclosing kernel functions within triply nested loops to iterate these kernel functions

Table X. Normalized Total Energy Consumptions of Randomly Generated TFGs with Setting C

Leakage contribution set to 10%						
number of MHP regions	method	dynamic	leakage[a]	leakage[b]	overhead	total
1	baseline	62.23%	9.83%	27.94%	0.00%	100.00%
	CADFA	62.35%	2.55%	29.14%	15.16%	109.19%
	MTPG	62.25%	8.33%	28.04%	0.34%	98.95%
2	baseline	58.90%	10.70%	30.40%	0.00%	100.00%
	CADFA	59.01%	2.57%	31.59%	12.74%	105.91%
	MTPG	58.93%	7.25%	30.60%	0.72%	97.50%
3	baseline	55.87%	11.49%	32.64%	0.00%	100.00%
	CADFA	55.97%	2.60%	33.89%	10.96%	103.42%
	MTPG	55.91%	6.11%	33.04%	1.19%	96.25%
4	baseline	55.85%	11.49%	32.65%	0.00%	100.00%
	CADFA	55.96%	2.67%	34.04%	10.05%	102.73%
	MTPG	55.89%	5.27%	33.19%	1.10%	95.45%
5	baseline	52.79%	12.29%	34.92%	0.00%	100.00%
	CADFA	52.90%	2.67%	36.23%	10.73%	102.53%
	MTPG	52.84%	4.00%	35.94%	2.34%	95.12%

Leakage contribution set to 30%						
number of MHP regions	method	dynamic	leakage[a]	leakage[b]	overhead	total
1	baseline	30.21%	18.15%	51.64%	0.00%	100.00%
	CADFA	30.26%	4.73%	53.87%	28.56%	117.43%
	MTPG	30.22%	15.40%	51.83%	0.62%	98.06%
2	baseline	27.28%	18.91%	53.80%	0.00%	100.00%
	CADFA	27.34%	4.57%	55.90%	22.91%	110.72%
	MTPG	27.30%	12.85%	54.15%	1.28%	95.57%
3	baseline	24.82%	19.55%	55.63%	0.00%	100.00%
	CADFA	24.86%	4.44%	57.75%	18.87%	105.93%
	MTPG	24.83%	10.40%	56.31%	2.03%	93.58%
4	baseline	24.86%	19.54%	55.60%	0.00%	100.00%
	CADFA	24.91%	4.55%	57.96%	17.20%	104.61%
	MTPG	24.88%	9.00%	56.49%	1.87%	92.24%
5	baseline	22.48%	20.16%	57.36%	0.00%	100.00%
	CADFA	22.53%	4.38%	59.51%	17.62%	104.04%
	MTPG	22.50%	6.57%	59.03%	3.84%	91.94%

[a]leakage energy consumed by power-gateable units.
[b]leakage energy consumed by other units.

in a given index range. Each thread computed the workload of a work group. The sources of OpenCL kernels were as follows: kernel *DCT, DwtHaar1D, FastWalshTransform, Histogram, MatrixTranspose, Permute, PrefixSum, RadixSort,* and *SimpleConvolution* are from AMD OpenCL SDK, while kernel *BP msg* is an OpenCL implementation of the BP application.

Figures 14 through 19 show our experimental results for BSP programs from OpenCL-based kernels. Figures 14 through 17 show the energy consumption normalized to the baseline case with no power-gating mechanism with different experimental parameters including leakage contribution and number of SMT threads. With a four-way SMT architecture, Figures 14 and 15 show the energy consumption on leakage contribution set to 10% and 30%, respectively. On leakage contribution set to 30%, the average reduction in total energy consumption was 10.09%, and was largest for *DCT* (10.84%) and the smallest for *Permute* (9.20%). On leakage contribution set to

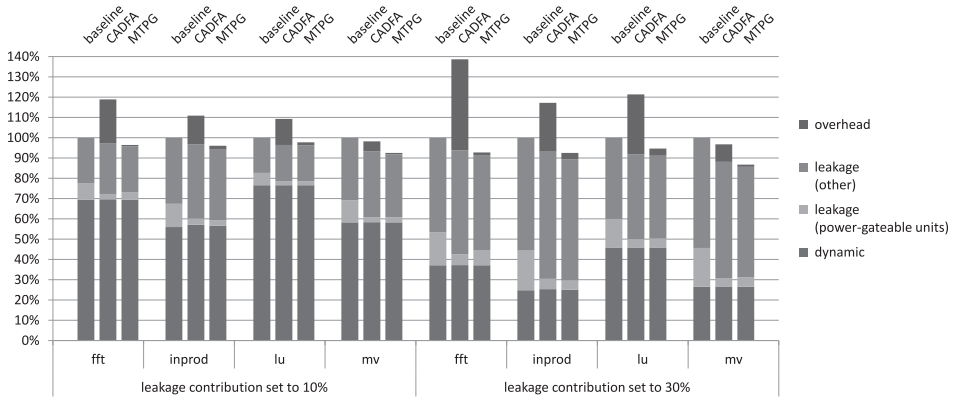

Fig. 13. Normalized total energy consumptions of BSP programs from BSPedupack.

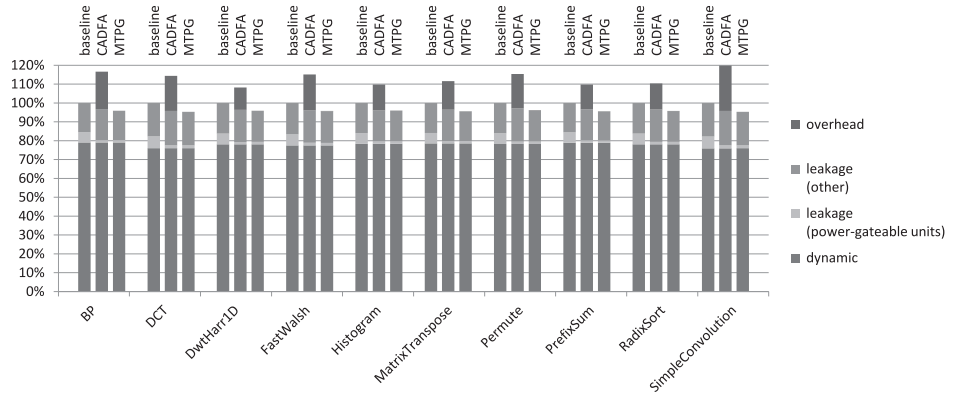

Fig. 14. Normalized total energy consumptions of BSP programs from OpenCL kernels on four-way SMT system with leakage contribution set to 10%.

10%, the average reduction in total energy consumption was 4.27%, and was largest for *DCT* (4.74%) and the smallest for *Permute* (3.86%). The energy breakdown of the BSP program from OpenCL kernels differs slightly from that for randomly generated D-BSP programs. On leakage contribution set to 30%, the leakage energies dissipated by power-gateable units were 3.16% and 3.19% in CADFA and MTPG, respectively. CADFA consumed nearly the same amount of leakage energy in power-gateable units as MTPG (about 99% energy consumption relative to MTPG), which explains why MTPG saves more energy in this setting than it does in randomly generated D-BSP programs. Figures 16 and 17 show the energy consumption in an experimental environment with eight-way SMT and the leakage contribution set to 10% and 30%, respectively. Experimental results show that our method could be applied to eight-way SMT architectures. As shown in Figure 17, while energy consumption of a system with CADFA grew, the system with MTPG successfully sustained the growing energy consumption and reduced 10% total energy on average.

The code sizes of OpenCL-based BSP programs relative to the baseline are shown in Figure 18. The comparison is based on the text section of a user program, excluding libraries and C runtime codes that could not be analyzed in our experimental environment. The average increases in code size due to the insertion of power-gating

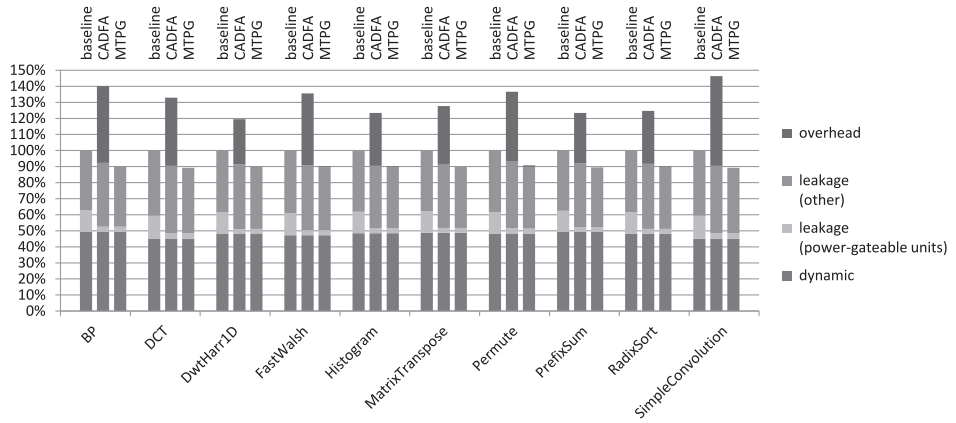

Fig. 15. Normalized total energy consumptions of BSP programs from OpenCL kernels on four-way SMT system with leakage contribution set to 30%.

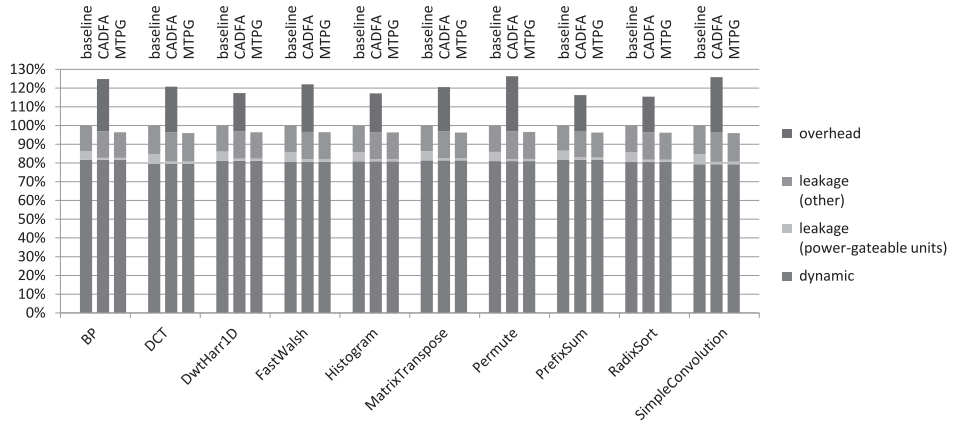

Fig. 16. Normalized total energy consumptions of BSP programs from OpenCL kernels on eight-way SMT system with leakage contribution set to 10%.

instructions were about 13% (ranging from 10.17% to 15.25%) and 3% (ranging from 1.45% to 5.26%) with CADFA and MTPG, respectively. The number of power-gating instructions of CADFA was reduced to about 89% using MTPGA, which reveals that MTPG efficiently inserts power-gating instructions for multithread programs.

Figure 19 shows the experimental results for OpenCL-based BSP programs with different configurations of the leakage contribution on a four-way SMT machine. MTPG reduced the total energy consumption from 4.28% to 18.54% for leakage contribution from 10% to 90%, respectively; in contrast, CADFA consumed more energy (from 1.13 to 1.56×) than the baseline case of no power-gating mechanism. The PPG and MTPG reduce leakage energy consumption by carefully managing the component status using predicated bits and appropriately inserting power-gating instructions. On the other hand, a large number of incorrect power-off instructions inserted by CADFA introduce many extra cycles while waiting for the internal powering on of components, and this deteriorates further as the leakage contribution increases. These observations indicate

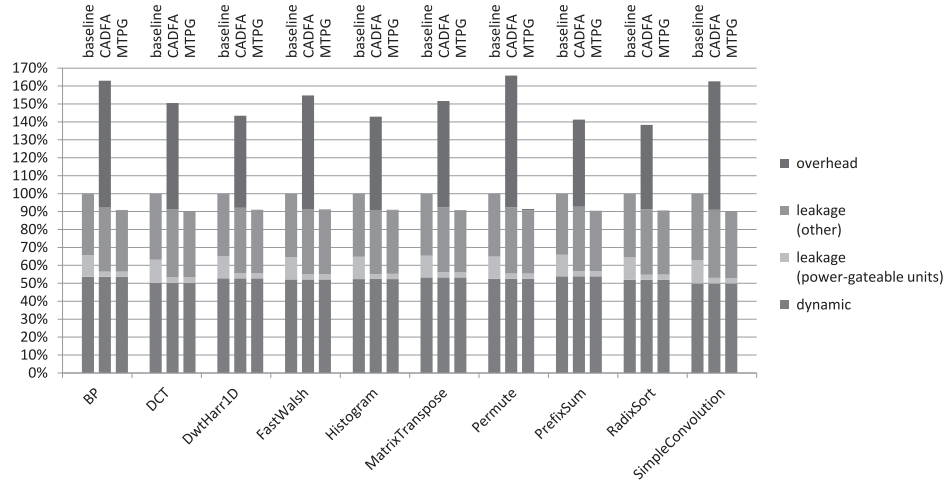

Fig. 17. Normalized total energy consumptions of BSP programs from OpenCL kernels on eight-way SMT system with leakage contribution set to 30%.

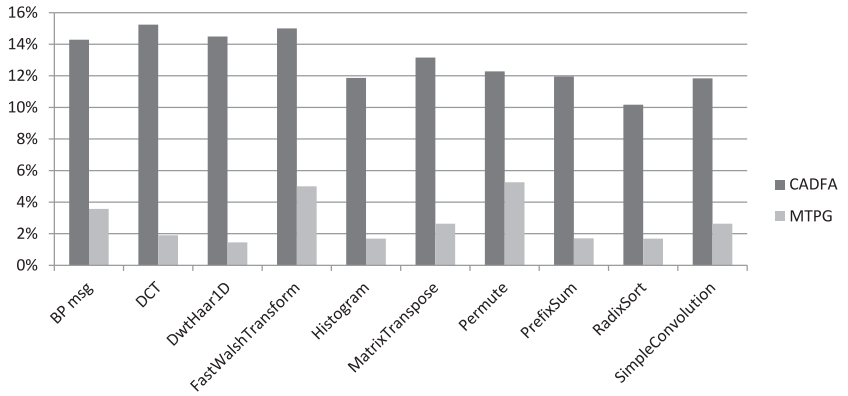

Fig. 18. Increases in code size.

that our technique is more effective than existing technologies at improving leakage control for BSP multithread programs.

7. DISCUSSION

In this section, we discuss the impact of latency and the capability to apply MTPGA on real hardware. Latency in processors affects execution time, which directly affects the result of power-gating optimization. Latencies in processors include pipelining latency and memory access latency. Pipelining latency is caused by pipeline hazards, where instructions are stalled because of structure hazards or nonresolved data dependencies. Memory access latency is caused by the memory hierarchy, such as cache miss.

Pipelining latency and memory access latency are both discussed in traditional power-gating analyses for single-thread environments such as CADFA [You et al. 2006] and sink-n-hoist [You et al. 2005, 2007]. These methods analyze component usage with regard to the shortest latency, which guarantees that leakage energy would be reduced in any case. CADFA is a conservative method because it estimates the saved leakage energy with the worst case of power gating [You et al. 2006]. The proposed MTPGA,

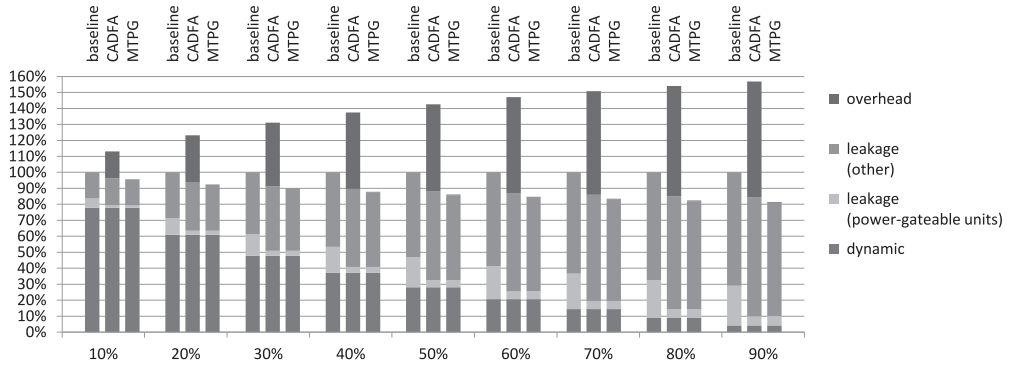

Fig. 19. Normalized energy consumptions for different leakage contributions.

based on CADFA, considers both pipelining latency and memory access latency as does CADFA. The latency of an instruction is considered with its minimal delay in our estimation; thus a multiply operation is considered with its shortest operation time and caches are considered perfect, which means that cache miss never occurs. Nevertheless, MTPGA also conservatively estimates the inactive period in an MHP region with the worst case, namely the minimal thread execution time among threads. With conservative estimation, the experimental results reveal that our method could save about 10% energy consumption on BSP programs (on leakage contribution set to 30%); the energy reduction can be further improved by using more precise analyses if the memory access time could be modeled at compile time. When the instruction fetching policy changes in SMT, our method is also applied because it estimates energy consumption with the worst case of concurrent threads, which guarantees that leakage energy would be reduced in any case. With a more precise performance analysis model for SMT, it is possible to further reduce the leakage energy. To apply our method to real systems or different processor architectures, one should update the estimation model with designated latency. Furthermore, one might be interested in incorporating varying latency analysis with the ILP estimation model [Li and Xue 2004] into a power model to improve the leakage energy savings.

Our method is capable of dealing with out-of-order execution with certain hardware support [You et al. 2006]. By dynamic scheduling techniques, superscalar processors fetch a bunch of instructions and issue these instructions concurrently with regard to data dependence among them, which may break the arrangement of power-gating operations in a thread inserted by a sequential compiler if the dependence between power-gating instructions and normal instructions is not properly considered. To ensure the inserted power-gating instructions are issued correctly, a power management controller could be implemented in chip to issue power-gating instructions at correct timing. The power management controller consists of a power direction buffer and component usage monitors. Once a power-gating instruction is decoded, the instruction dispatcher dispatches the instruction to the power direction buffer of the power management controller. The power management controller is capable of knowing the component usage by monitoring instructions at reservation stations. When the power management controller detects that all instructions using the component are completed, it would issue the power-gating instructions in the power direction buffer and turn off the component according to a power directive. Finally, the power management controller removes the power-gating instructions from the power-gating direction buffer. In this regard, the situation where an instruction finds its function unit turned off can be avoided, meaning our approach can be applied to out-of-order machines.

The key idea of this study is to save leakage energy of shared execution resources in a system by special hardware support and a compiler analysis method. In this work, we focus on examining the energy efficiency of MTPG on SMT-based systems with our proposed mechanisms, but the method is not limited to SMT-based systems. Rather, it is capable of being applied to systems with shared execution resources. OpenCL is a programming model for GPUs, where several features are similar to the hierarchical BSP model, such as multithread, global synchronization operation, and private memory in the memory hierarchy. It would be a possible direction for future research to apply our method on GPU architectures.

8. CONCLUSION

This article has presented a foundation framework for compilation optimization that reduces the power consumption on SMT architectures. It has also presented PPG operations for improving the energy management of multithread programs in hierarchical BSP models. Based on a multithread component analysis with dataflow equations, our MTPGA framework estimates the energy usage of multithread programs and inserts PPG operations as power controls for energy management. Our preliminary experimental results on a system with leakage contribution set to 30% show that using a system with PPG support and using the MTPGA method reduced the total energy consumption by an average of 10.09% for BSP programs and by up to 10.49% for D-BSP programs relative to the system without a power-gating mechanism, and reduced the total energy consumption by an average of 4.27% for BSP programs and by up to 6.68% for D-BSP programs on a system with leakage contribution set to 10%, demonstrating that our mechanisms are effective in reducing the leakage power in hierarchical BSP multithread environments.

REFERENCES

R. Barik. 2005. Efficient computation of may-happen-in-parallel information for concurrent Java programs. In *Proceedings of the 18th International Conference on Languages and Compilers for Parallel Computing (LCPC'05).* Lecture Notes in Computer Science, vol. 4339, Springer, 152–169.

N. Bellas, I. N. Hajj, and C. D. Polychronopoulos. 2000. Architectural and compiler techniques for energy reduction in high-performance microprocessors. *IEEE Trans. VLSI* 8, 3, 317–326.

R. H. Bisseling. 2004. *Parallel Scientific Computation: A Structured Approach using BSP and MPI*. Oxford University Press.

J. A. Butts and G. S. Sohi. 2000. A static power model for architects. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'00).* 191–201.

D. Callahan and J. Sublok. 1989. Static analysis of low level synchronization. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging (PADD'89).* 100–111.

H. Cha and D. Lee. 2001. H-BSP: A hierarchical bsp computation model. *J. Supercomput.* 18, 2, 179–200.

A. P. Chandrakasan, S. Sheng, and R. W. Brodersen. 1992. Low-power cmos digital design. *IEEE J. Solid-State Circ.* 27, 4, 473–484.

J.-M. Chang and M. Pedram. 1995. Register allocation and binding for low power. In *Proceedings of the Design Automation Conference (DAC'95).* 29–35.

D. Cordeiro, G. Mounie, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner. 2010. Random graph generation for scheduling simulations. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools'10).* 60:1–60:10.

S. Dropsho, V. Kursun, D. H. Albonesi, S. Dwarkadas, and E. G. Friedman. 2002. Managing static leakage energy in microprocessor functional units. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO'02).* 321–332.

E. Duesterwald and M. L. Soffa. 1991. Concurrency analysis in the presence of procedures using a data-flow framework. In *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV'91).* 36–48.

J. Goodacre. 2011. Understanding what those 250 million transistors are doing. In *Proceedings of the 11th International Forum on Embedded MPSoC and Multicore (MPSoC'11).*

M. Horowitz, T. Indermaur, and R. Gonzalez. 1994. Low-power digital design. In *Proceedings of the IEEE Symposium on Low Power Electronics.* 8–11.

P. Y. T. Hsu and E. S. Davidson. 1986. Highly concurrent scalar processing. In *Proceedings of the 13th Annual International Symposium on Computer Architecture (ISCA'86).* 386–395.

Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose. 2004. Microarchitectural techniques for power gating of execution units. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'04).* 32–37.

J. T. Kao and A. P. Chandrakasan. 2000. Dual-threshold voltage techniques for low-power digital circuits. *IEEE J. Solid-State Circ.* 35, 7, 1009–1018.

C. W. Kessler. 2000. NestStep: Nested parallelism and virtual shared memory for the bsp model. *J. Supercomput.* 17, 3, 245–262.

C. Lee, J. K. Lee, T.-T. Hwang, and S.-C. Tsai. 2003. Compiler optimizations on vliw instruction scheduling for low power. *ACM Trans. Des. Autom. Electron. Syst.* 8, 2, 252–268.

J. Lee, J. Kim, S. Seo, S. Kim, J. Park, H. Kim, T. T. Dao, Y. Cho, S. J. Seo, S. H. Lee, S. M. Cho, H. J. Song, S.-B. Suh, and J.-D. Choi. 2010. An opencl framework for heterogeneous multicores with local memory. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10).* ACM Press, New York, 193–204.

J. Lee, J. M. Youn, D. Cho, and Y. Paek. 2013. Reducing instruction bit-width for low-power vliw architectures. *ACM Trans. Des. Autom. Electron. Syst.* 18, 2, 25:1–25:32.

M. T.-C. Lee, V. Tiwari, S. Malik, and M. Fujita. 1997. Power analysis and minimization techniques for embedded dsp software. *IEEE Trans. VLSI Syst.* 5, 1, 123–133.

L. Li and C. Verbrugge. 2004. A practical mhp information analysis for concurrent java programs. In *Proceedings of the 17th International Conference on Languages and Compilers for Parallel Computing (LCPC'04).* Lecture Notes in Computer Science, vol. 3602, Springer, 194–208.

L. Li and J. Xue. 2004. A trace-based binary compilation framework for energy-aware computing. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04).* ACM Press, New York, 95–106.

S. P. Masticola and B. G. Ryder. 1993. Non-concurrency analysis. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'93).* 129–138.

W. F. McColl. 1996. Universal computing. In *Proceedings of the 2nd International Euro-Par Conference on Parallel Processing (Euro-Par'96).* Lecture Notes in Computer Science, vol. 1123, Springer, 25–36.

G. Naumovich and G. S. Avrunin. 1998. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel for rendezvous-based concurrent programs. In *Proceedings of the 6th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'98).* 24–34.

G. Naumovich, G. S. Avrunin, and L. A. Clarke. 1999. An efficient algorithm for computing mhp information for concurrent java programs. In *Proceedings of the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'99).* Lecture Notes in Computer Science, vol. 1687, Springer, 338–354.

G. Ramalingam. 2000. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.* 22, 2, 416–430.

S. Rele, S. Pande, S. Onder, and R. Gupta. 2002. Optimizing static power dissipation by functional units in superscalar processors. In *Proceedings of the 11th International Conference on Compiler Construction (CC'02).* 261–275.

S. Rusu, S. Tam, H. Muljono, D. Ayers, J. Chang, B. Cherkauer, J. Stinson, J. Benoit, R. Varada, J. Leung, et al. 2007. A 65-nm dual-core multithreaded xeon® processor with 16-mb l3 cache. *IEEE J. Solid-State Circ.* 42, 1, 17–25.

C.-L. Su and A. M. Despain. 1995. Cache designs for energy efficiency. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences (HICSS'95).* 306–315.

R. N. Taylor. 1983. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica* 19, 57–84.

V. Tiwari, R. Donnelly, S. Malik, and R. Gonzalez. 1997. Dynamic power management for microprocessors: A case study. In *Proceedings of the International Conference on VLSI Design (VLSID'97).* 185–192.

V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. Patel, and F. Baez. 1998. Reducing power in high-performance microprocessors. In *Proceedings of the 35th Annual Design Automation Conference (DAC'98).* 732–737.

P. D. L. Torre and C. P. Kruskal. 1996. Submachine locality in the bulk synchronous setting (extended abstract). In *Proceedings of the 2nd International Euro-Par Conference on Parallel Processing (Euro-Par'96).* Vol. 2. Springer, 352–358.

L. G. Valiant. 1990. A bridging model for parallel computation. *Comm. ACM* 33, 8, 103–111.

L. G. Valiant. 2008. A bridging model for multi-core computing. In *Proceedings of the 16th Annual European Symposium on Algorithms (ESA'08)*. 13–28.

L. G. Valiant. 2011. A bridging model for multi-core computing. *J. Comput. Syst. Sci.* 77, 1, 154–166.

H. Yang, R. Govindarajan, G. R. Gao, G. Cai, and Z. Hu. 2002. Exploiting schedule slacks for rate-optimal power-minimum software pipelining. In *Proceedings of the 3rd Workshop on Compilers and Operating Systems for Low Power (COLP'02)*.

Y.-P. You, C.-W. Huang, and J. K. Lee. 2005. A sink-n-hoist framework for leakage power reduction. In *Proceedings of the ACM International Conference on Embedded Software (EMSOFT'05)*. 83–94.

Y.-P. You, C.-W. Huang, and J. K. Lee. 2007. Compilation for compact power-gating controls. *ACM Trans. Des. Autom. Electron. Syst.* 12, 4.

Y.-P. You, C. Lee, and J. K. Lee. 2002. Compiler analysis and supports for leakage power reduction on microprocessors. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC'02)*. Lecture Notes in Computer Science, vol. 2481, Springer, 63–73.

Y.-P. You, C. Lee, and J. K. Lee. 2006. Compilers for leakage power reduction. *ACM Trans. Des. Autom. Electron. Syst.* 11, 1, 147–164.

W. Zhang, M. T. Kandemir, N. Vijaykrishnan, M. J. Irwin, and V. De. 2003. Compiler support for reducing leakage energy consumption. In *Proceedings of the 6th Design Automation and Test in Europe Conference (DATE'03)*. 1146–1147.

V. Zivojnovic, J. M. Velarde, and C. Schlager. 1994. DSPstone: A dsp-oriented benchmarking methodology. In *Proceedings of 5th International Conference on Signal Processing Applications and Technology*.

Received October 2013; revised August 2014; accepted September 2014